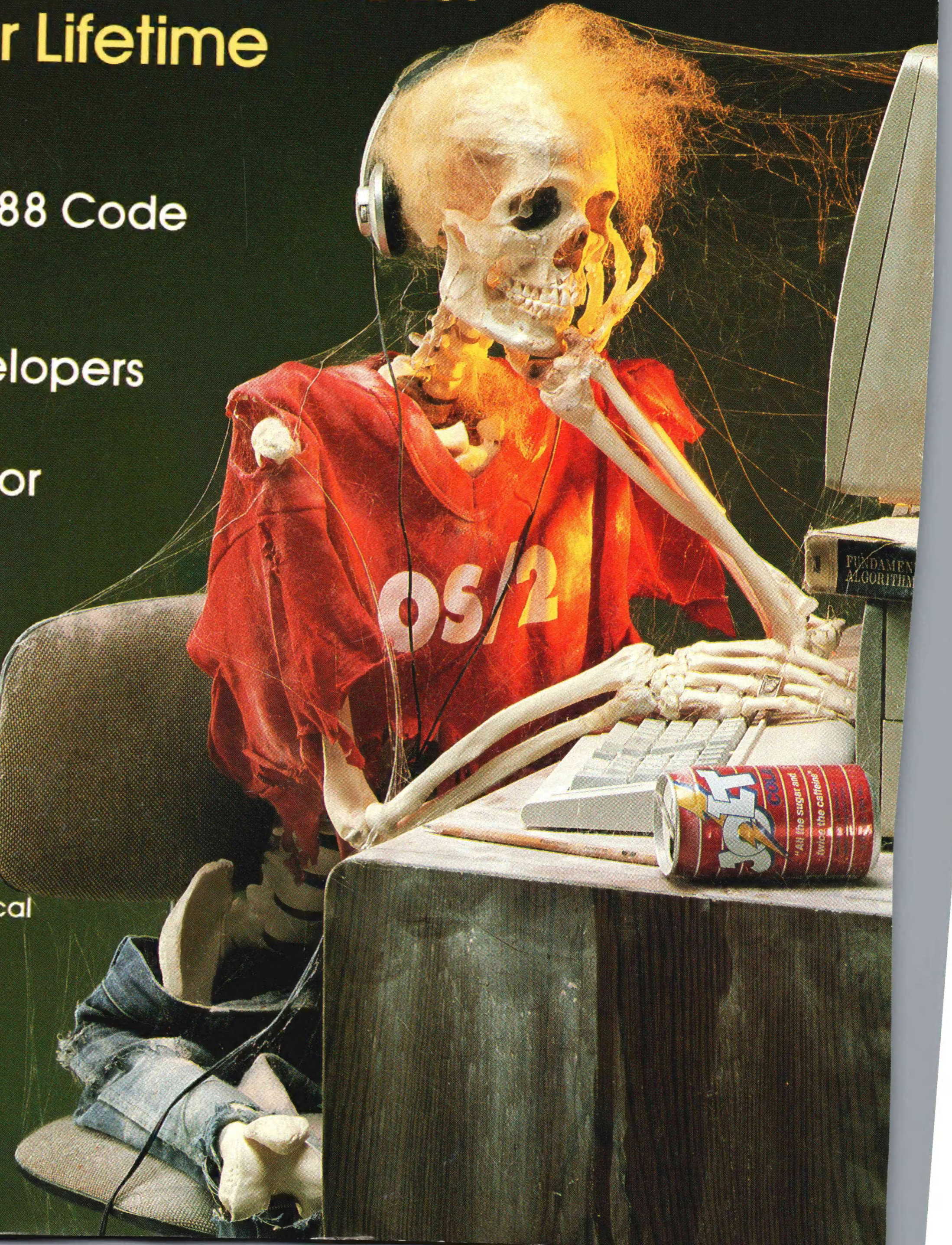
#129 JULY 1987

2.95 (3.95 CANADA)

Dr. Dobb's Journal of Software Tools

FOR THE PROFESSIONAL PROGRAMMER

386 DEVELOPMENT TOOLS: Within Your Lifetime



Optimizing 8088 Code

Rules for Software Developers

Workarounds for PROLOG

Curses for MS-DOS

Languages:
A Curses Package in C
Forth: Rules to Code By
Multitasking in Turbo Pascal
Inside a LISP Machine
BASIC Data Types

07
38351 16562 8

*"Turbo C does look like What We've All Been Waiting For: a full-featured compiler that produces excellent code in an unbelievable hurry . . .
. . . moves into a class all its own among full-featured C compilers . . .
. . . Turbo C is indeed for the serious developer . . . One heck of a buy—at any price.*

Michael Abrash
Programmer's Journal"

*"Borland International's Turbo Pascal, Turbo Basic and Turbo Prolog automatically identify themselves, by virtue of their "Turbo" forenames, as superior language products with a common programming environment. The apellation also means, to many PC users, a "must have" language . . .
. . .To us, Turbo C looks like a coup for Borland.*

Garry Ray, PC Week"

Reflex: The Database Manager

With Reflex, we brought new eyes and understanding to spreadsheets with unheard-of graphics, charts, plots and analysis. And just as we do in our Language products, we've added our Reflex Workshop™, a "business toolbox" which gives you everything you need to set up and run more than 20 different kinds of business.

"Reflex actually improves on 1-2-3 in several respects.

William Casey, PC Tech Journal"

"Reflex and Reflex Workshop may be the best bargain in software today. Highly recommended.

Jerry Pournelle, Byte"

Eureka™: The equation solver for Scientists, Engineers, Students and Professionals

Eureka is as much a step forward in equation solving as spreadsheets have been for accounting and planning. Using Eureka with your PC gives you high performance in desktop engineering. A must!

"Eureka is fast, and simple to use.

Michael Miller, InfoWorld"

Sprint®: Bringing all that technology together

Word Processors? There are a lot of good ones around. WordPerfect™, WordStar® 4.0, and MicroSoft® Word are good products. But to compete, we had to build a better product. And using some of the technology we pioneered in our fast compilers, we set out to build the best professional word processor ever written.

We spent a lot of time listening to word processor users. Finding out what they liked, and what they really didn't like about the programs they were using.

We looked at all the features of current word processors and figured out how to make them better. That's how we designed Sprint: The Professional Word Processor, which you'll soon discover is faster, easier, smarter than the rest, and probably the most powerful word processor ever written. Sprint is well worth the wait—and you won't be waiting long.

"Sprint: The Word Processor is a many-splendored package.

Ken Greenberg, PC World"

Why it's good for you that Borland's in the business of Business

At Borland, small is actually beautiful. Either Joy or Anne or Dyane will answer the phone—but don't get fresh, because they're all happily married. Call Borland and you talk to an actual human being who doesn't give you the big corporate runaround, but the right extension number. You'll get help, whether it's tech support, questions about new products, or whatever. (We've always had a 60-day money-back guarantee, so we don't get too many calls about that.)

When you're small, you try harder, so what we've done is just that.

BI-1126BW

Borland has to be a whole lot better because our competition is a whole lot bigger

At Borland we're competing with giants like Microsoft and Lotus, so we have to try harder. Out-think them, out-smart them, out-pace them, out-perform them—because we obviously can't out-spend them.

What that means to you—which is all that really matters—is that you now have access to the world's fastest compilers, because we were forced to invent them to compete with the giants. We may be smaller than they are, but we're absolutely, positively not slower than they are.

In the long run what really counts is technological excellence.

BI-1126BW

Our technology is so advanced, it's easy to use

With Turbo Pascal® we invented fast compilers, and followed that technological breakthrough with Turbo Prolog®, then Turbo Basic®, then Turbo C.® Building these fast compilers is not an easy task. It was and is "the little guy" taking a giant step. And having transformed Languages with our new superfast compiler technology, we've turned the same power loose onto our Business products like SideKick® and Reflex®:The Database Manager.

> *"SideKick continues to influence not only the utilities market but the entire software industry."*
> William Urschel, PC World

When Borland was founded 4 years ago, the software industry technology level was about C– on a scale of A to F. Which was and is a perfect opportunity for a technology-driven company like Borland. Call us "techies," but we're developers, technicians, tinkerers who know how to make programs run faster, do more, be more and let you fully use the hardware power you've paid for.

At Borland, Price is one thing, but Technology is the main thing

There's no sense in the "price being right" if the product is wrong. A useless product is something you can't give away. Technical excellence and superiority always comes ahead of Price. Always has. Always will at Borland. Our technical leadership began when we invented fast compilers and hasn't been matched since.

Turbo Pascal: The world-wide standard

Pascal was asleep before we transformed it with a technical shot in the arm. Our unique ability to create spectacularly fast compilers was the driving force behind Turbo Pascal's worldwide success.

> *"For the IBM PC, the benchmark Pascal compiler is undoubtedly Borland International's Turbo Pascal."*
> Garry Ray, PC WEEK

Turbo Basic: BASIC raised to a new power

We've raised BASIC from the dead with our recent high-speed Turbo Basic. Of course, Microsoft will try to sell you their "QuickBASIC," but we think you're interested in "fast," not "quick." Because we're a smaller company, we had to make Turbo Basic the "best BASIC development environment ever written." Otherwise, we'd be out of business. We try harder.

> *"Borland International's Turbo Basic is unquestionably an outstanding software product."*
> Giovanni Perrone, PC WEEK

> *"Turbo Basic is a compiled BASIC. This gives it execution speeds that leave standard interpretive BASICs like BASICA and GW BASIC in the dust."*
> William Zachmann
> COMPUTERWORLD

Turbo Prolog: The natural language of Artificial Intelligence

Turbo-charging Prolog was an enormous challenge. Creating a development environment on an ordinary PC that would rival those found on dedicated AI workstations like Sun and Apollo was deemed impossible. Enter Turbo Prolog. Exit the rest.

> *"Turbo Prolog has one of the most powerful user interfaces ever seen in a software development system."*
> Tom Swan
> Programmer's Journal

> *"If you're at all interested in artificial intelligence, databases, expert systems, or new ways of thinking about programming, by all means plunk down your $100 and buy a copy of Turbo Prolog."*
> Bruce Webster, Byte

Turbo C: Perhaps the most powerful professional development environment ever written

We've brought tomorrow's technology to Turbo C, which is so fast that it makes the rest look like dead Cs. We've given Turbo C a revolutionary user interface making it a wonderful productivity booster. And in keeping with our commitment to open architectures, we are even offering our users the opportunity to license the source code to Turbo C's run-time library.

BI-1126BW

Borland Software: Technical superiority, innovation and high performance

NEW

Turbo Pascal Numerical Methods Toolbox™
High-performance, state-of-the-art numerical routines for Turbo Pascal® programmers. A must for Scientists, Students and Engineers!
Only $99.95

NEW

Turbo C®
It's the fastest, most powerful, full-featured C compiler at any price. It's the one you've been waiting for!
Only $99.95

NEW

Turbo Prolog Toolbox™
The new collection of efficient, "smart" tools for use with our highly-acclaimed Turbo Prolog,® the natural language of Artificial Intelligence
Only $99.95

NEW

Turbo Basic®
It's the high-speed BASIC you'd expect from Borland. A complete development environment including an amazingly fast compiler, interactive editor and trace debugging system. Compatible with BASICA
Only $99.95

NEW

Eureka: The Solver™
The innovative, easy-to-use equation solver. A must for Scientists, Engineers, Architects, Financial Analysts, Students, Educators and other Professionals
Only $99.95

BORLAND
INTERNATIONAL

For the dealer nearest you or to order by phone call
(800) 255-8008
in CA (800) 742-1133
In Canada (800) 237-1136

4585 SCOTTS VALLEY DRIVE SCOTTS VALLEY, CA 95066 (408) 438-8400 TELEX: 172373

CIRCLE 161 ON READER SERVICE CARD

PC MAGAZINE EDITOR'S CHOICE

. . . one package stands out as the best support available for Turbo Pascal programmers: ***Blaise Computing's Turbo Power Tools.*** *This definitive set of prewritten Pascal functions and procedures will make the life of any programmer—from the beginner to the hard-core professional—easier and more productive.*

Power Tools Plus™

Window Routines. Memory Resident Routines. Routinely.

ANOTHER PLUS FROM BLAISE COMPUTING

The best just got better! Turbo POWER TOOLS, acclaimed as the best programmer support package for Turbo Pascal, now has even more functions, more detailed documentation and more sample programs.

NO SECRETS

Turbo POWER TOOLS PLUS is crafted so that the source is efficient, readable and easy to modify. We don't keep secrets! We tell you exactly how windows are managed, how interrupt service routines can be written in Turbo Pascal, and how to write memory resident programs that can even access the disk. Maybe you've heard of some undocumented DOS features that resident programs use to weave their magic. Turbo POWER TOOLS PLUS documents these features and lets you make your own magic!

Here's just part of the PLUS in Turbo POWER TOOLS PLUS:

- **WINDOWS** that are stackable, removable, with optional borders and a cursor memory.
- **FAST DIRECT VIDEO ACCESS** for efficiency.
- **SCREEN HANDLING** including multiple monitor and EGA 43-line support.
- **POP-UP MENUS** which are flexible, efficient and easy to use, giving your applications that polished look.
- **INTERRUPT SERVICE ROUTINES** that can be written in Turbo Pascal without the need for assembly language or inline code.
- **INTERVENTION CODE** lets you develop memory resident applications that can take full advantage of DOS capabilities. With simple procedure calls, you can "schedule" a Turbo Pascal procedure to execute either when a "hot key" is pressed, or at a specified time.
- **PROGRAM CONTROL ROUTINES** allow you to run other programs from Turbo Pascal, and even execute DOS commands.
- **MEMORY MANAGEMENT** allows you to monitor, allocate and free DOS-controlled memory.
- **DIRECTORY AND FILE HANDLING** support to let you take advantage of the newer features of DOS including networking.
- **STRING** procedures allowing powerful translation and conversion capabilities.
- **FULL SOURCE CODE** for all included routines, sample programs and utilities.
- **DOCUMENTATION, TECHNICAL SUPPORT** and attention to detail that have distinguished Blaise Computing over the years.

Turbo POWER TOOLS PLUS *supports Turbo Pascal Version 2.0 and later and is just* ***$99.95.***

Another quality product from Blaise Computing: ***Turbo ASYNCH PLUS™***

A new package which provides the crucial core of hardware interrupt support needed to build applications that communicate. ASYNCH PLUS offers simultaneous buffered input and output to both COM ports at speeds up to 9600 baud. The XON/XOFF protocol is supported. Now it also includes the "XMODEM" file-transfer protocol and support for Hayes compatible modems.

The underlying functions of Turbo ASYNCH PLUS are carefully crafted in assembler for efficiency and drive the UART and programmable interrupt controller chips directly. These functions, installed as a runtime resident system, require just 3.2K bytes. The high level function are all written in Turbo Pascal in the same style and format as Turbo POWER TOOLS PLUS. All source code is included for just $99.95.

BLAISE COMPUTING INC.

2560 Ninth Street, Suite 316 Berkeley, CA 94710 (415) 540-5441

ORDER TOLL-FREE 800-227-8087

Calif. residents call (415) 540-5441

YES, send me the PLUS I need! Enclosed is $________ for
☐ Turbo POWER TOOLS PLUS ☐ Turbo ASYNCH PLUS ☐
OTHER ________. (CA residents add 6½% Sales Tax. All domestic orders add $10.00 for Federal Express shipping.)
Name: ________ Phone: (____) ________
Shipping Address: ________ Zip: ________
City: ________ State: ________
VISA or MC #: ________ Exp. Date: ________

CIRCLE 159 ON READER SERVICE CARD

CONTENTS

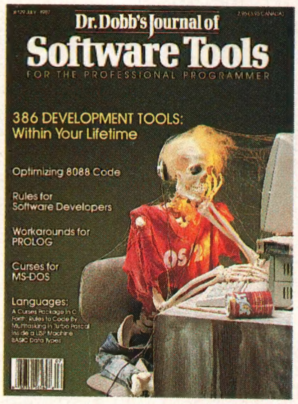
VOLUME 12, ISSUE 7

ARTICLES

COLUMNS

FORUM

PROGRAMMER'S SERVICES

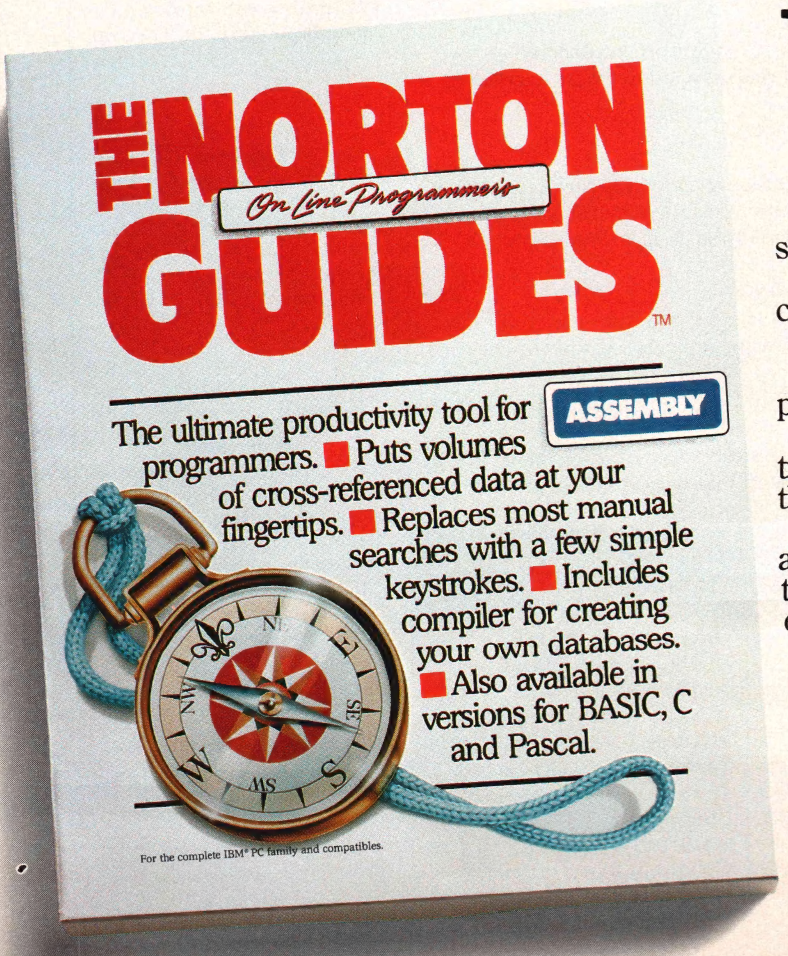


About the Cover
Most of the props for this shot, with the probable exception of the skeleton, are items that any programmer may have lying around. The real killer, of course, is the missing software. (Special thanks to Donald Knuth for an outstanding reference and to Dennis Brothers for the brass rat.)

This Issue
Are you suffering from those wait-another-year-for-the-gang-in-Redmond blues? This month both Richard Relph and Ray Duncan examine some of 386 development tools already available.

Next Issue
Next month's issue has a special emphasis on tools for C programmers. The lead article examines both the pleasant and troublesome surprises with the coming ANSI standard C and offers some prescriptions and preventive measures.

Peter Norton new programmi who hate



Nobody ever said programming PCs was supposed to be easy.

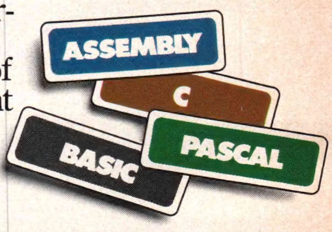
But does it have to be tedious and time-consuming, too?

Not any more.

Not since the arrival of the remarkable new program on the left.

Which is designed to save you most of the time you're currently spending searching through the books and manuals on the shelf above.

The Norton On-Line Programmer's Guides™ are a quartet of pop-up reference packages that do the same things in four different languages.



Each package consists of two parts: A memory-resident instant access program. And a comprehensive, cross-referenced database crammed with just about everything you need to know to program in your favorite language.

And when we say everything, we mean everything.

Everything from information about language

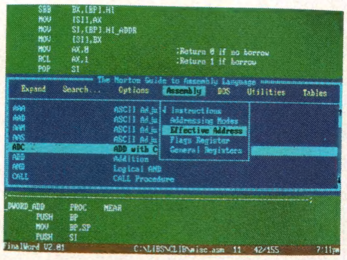
Designed for the IBM® PC, PC-AT and DOS compatibles. Available at most software

announces a
ng tool for people
e manual labor.

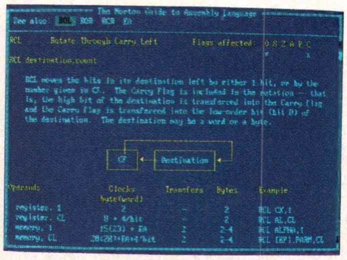
syntax to a variety of tables, including ASCII characters, line drawing characters, error messages, memory usage maps, important data structures and more.

How much more?

Well, the databases for BASIC, C and Pascal give you detailed listings of all built-in and library functions.



A Guides reference summary screen (shown in blue) pops up on top of the program you're working on (shown in green).



Summary data expands on command into extensive detail. And you can select from a wide variety of information.

While the Assembly database delivers a complete collection of DOS service calls, interrupts and ROM BIOS routines.

You can, of course, find most of this information in the books and manuals on our shelf.

But Peter Norton—who's written a few books himself—figured you'd rather have it on your screen.

In seconds.

In full-screen or moveable half-screen mode.

Popping up right next to your work. Right where you need it.

This, you're probably thinking, is precisely the kind of thinking that produced the classic Norton Utilities.™

And you're right.

But even Peter Norton can't think of everything.

Which is why there's a built-in compiler for creating databases of your own.

And why all Guides databases are compatible with the instant access program in your original package.

So you can add more languages without spending a lot more money.

To get more information, call your dealer or Peter Norton Computing.

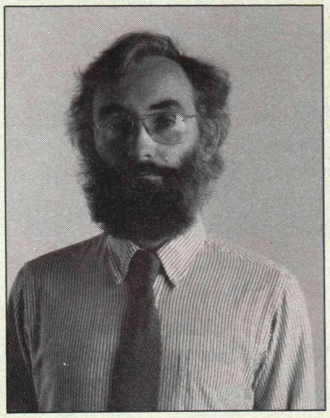
And ask for some guidance.

Peter Norton
COMPUTING

dealers, or direct from Peter Norton Computing, Inc., 2210 Wilshire Blvd., #186, Santa Monica, CA 90403. 213-453-2361, Fax 213-453-6398, MCI Mail: PNCI ©1987 Peter Norton Computing

CIRCLE 87 ON READER SERVICE CARD

EDITORIAL



This month's cover whimsically illustrates the frustration some programmers have expressed over the delivery by IBM of 286- and 386-based machines designed to run Microsoft's OS/2 multitasking operating system when there will not be a commercially available OS/2 this year.

The announced OS/2 timetable is: (1) Software Development Kits in developers' hands by next month, including kernel software, development languages and tools, and specs for the Windows Presentation Manager and LAN Manager; (2) Windows Presentation Manager and LAN Manager shipped as SDK updates by November; (3) OS/2 kernel shipped to OEMs by the end of the year; and (4) delivery dates for Windows Presentation Manager and LAN Manager to OEMs and anything at all to end users to be announced. That's for the 286/386 version of OS/2; no timetable has been announced for the later release that will take advantage of all the capabilities of the 386.

How you view that schedule depends, perhaps, on whether you view OS/2 as the operating system of the 1990s or as Microsoft's response to the AT.

Last year I watched a number of technical journalists racing with Bill Gates to write a program to perform a simple task. Storm the Gates, the contest was called, and Bill won it, using QuickBASIC, hands down. As I see it, Bill's current challenge is to get OS/2 out before Don Knuth completes *The Art of Computer Programming*, and that may be a tougher challenge.

Meanwhile, you don't have to wait for OS/2 to begin writing software for the 386 machines. Our lead article this month details some ways in which you can get started today.

And, meanwhile—well, there are a lot of meanwhiles. Digital Research has reorganized, with a new CEO and with founder Gary Kildall taking a more active role after a hiatus of three years. DRI has ported its Concurrent DOS to the 386 and also has a real-time OS, FLEXOS 386, in beta test now. DRI's products have been a lot better than the company's marketing in the past; we'll see if that continues. THEOS software has ported its multiuser, multitasking THEOS operating system to the 386. AT&T's Unix and Microsoft's Xenix are converging to one product for the 386. Microsoft perceives the 386 multiuser market to be small, but there are a lot of single-user Unix workstations that may be replaced by 386 machines, and there will be pressure to put Unix on those 386s.

Still, the most common operating system on 386 machines in 1988 will probably be MS-DOS 3.x.

None of this should be taken as evidence that we don't think OS/2 will be important for 286 and 386 machines. We are sending people to the developer conferences and are even setting up an OS/2 development lab in our offices. *DDJ* editors who have worked with OS/2 say it is an impressive piece of work. We do think OS/2 will be important—when it arrives.

In my remaining space I'd like to introduce *DDJ*'s new editor, Tyler Sperry. Tyler meets all my criteria for a good editor: good technical credentials, an appreciation for the effective use of English, willingness to work long hours for low pay. . . . Tyler is every bit as wonderful as OS/2, and he's here now. In fact, you can meet him on page eight.

Michael Swaine

Michael Swaine
editor-in-chief

Dr. Dobb's Journal of Software Tools
FOR THE PROFESSIONAL PROGRAMMER

Editorial
Editor-in-Chief *Michael Swaine*
Editor *Tyler Sperry*
Managing Editor *Vince Leone*
Assistant Editors *Sara Noah Ruddy*, *Levi Thomas*
Technical Editor *Allen Holub*
Consulting Editor *Nick Turner*
Contributing Editors *Ray Duncan*, *Michael Ham*, *Bela Lubkin*, *Namir Shammas*, *Ernest R. Tello*
Copy Editor *Rhoda Simmons*
Production
Production Manager *Bob Wynne*
Art Director *Michael Hollister*
Assoc. Art Director *Joe Sikoryak*
Technical Illustrator *Frank Pollifrone*
Cover Photographer *Michael Carr*
Circulation
Circulation Director *Maureen Kaminski*
Newsstand Sales Mgr. *Stephanie Ericson*
Book Marketing Mgr. *Jane Sharninghouse*
Circulation Coordinator *Kathleen Shay*
Administration
Finance Director *Kate Wheat*
Business Manager *Betty Trickett*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accts. Receivable Supv. *Laura Di Lazzaro*
Advertising Director
Ferris Ferdon (415) 366-3600
Account Managers
see page 117
Promotions/Srvcs. Mgr. *Anna Kittleson*
Advertising Coordinator *Charles Shively*
Associate Publisher
Michael Swaine
Assistant *Sara Noah Ruddy*

Dr. Dobb's Journal of Software Tools (USPS 307690) is published monthly by M&T Publishing Inc., 501 Galveston Dr., Redwood City, CA 94063; (415) 366-3600. Second-class postage paid at Redwood City and at additional entry points. *DDJ* is published under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a nonprofit corporation.

Article Submissions: Send manuscripts and disk (with article and listings) to the Editor.

***DDJ* on CompuServe:** Type GO DDJ

Address Correction Requested: Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 27809, San Diego, CA 92128. **ISSN 0888-3076**

Customer Service: For subscription problems call: outside CA (800) 321-3333; in CA (619) 485-9623 or 566-6947. For book/software order problems call (415) 366-3600.

Subscriptions: $29.97 per 1 year; $56.97 for 2 years. Canada and Mexico add $27 per year airmail or $10 per year surface. All other countries add $27 per year airmail. Foreign subscriptions must be prepaid in U.S. funds drawn on a U.S. bank. For foreign subscriptions, TELEX: 752-351.

Foreign Newsstand Distributor: Worldwide Media Service Inc., 386 Park Ave. South, New York, NY 10016; (212) 686-1520 TELEX: 620430 (WUI).

M&T Publishing Inc.
Chairman of the Board *Otmar Weber*
Director *C. F. von Quadt*
President and Publisher *Laird Foshay*

Our thanks to NASA for supplying this computer enhanced ultraviolet photo taken by Skylab IV of a solar prominence reaching out 350,000 miles above the sun's surface

Genius Begins With A Great Idea ...

But The Idea Is Just The Beginning

What follows is the time consuming task of giving form and function to the idea.

That's why we concentrate on building into our software development systems functions and features that help you develop your software ideas in less time and with less effort.

We've started 1987 by releasing new versions of our MS-DOS, Macintosh, Amiga, ROM, and Apple // C development systems. Each system is packed with new features, impressive performance, and a little bit more genius.

Aztec C86 4.1

New PC/MS-DOS • CP/M-86 • ROM

Superior performance, a powerful new array of features and utilities, and pricing that is unmatched make the new Aztec C86 the first choice of serious software developers.

Aztec C86-p Professional System $199
• optimized C with near, far, huge, small, and large memory + Inline assembler + Inline 8087/80287 + ANSI support + Fast Float (32 bit) + optimization options • Manx Aztec 8086/80x86 macro assembler • Aztec overlay linker (large/small model) • source level debugger • object librarian • 3.x file sharing & locking • comprehensive libraries of UNIX, DOS, Screen, Graphics, and special run time routines.

Aztec C86-d Developer System $299
• includes all of Aztec C86-p • Unix utilities make, diff, grep • vi editor • 6 + memory models • Profiler.

Aztec C86-c Commercial System. $499
• includes all of Aztec C86-d • Source for library routines • ROM Support • CP/M-86 support • One year of updates.

Aztec C86 Third Party Software

A large array of support software is available for Aztec C86. Call or write for information. The following is a list of the most requested products: Essential Graphics • C Essentials • C Utility Library • Greenleaf Com. • Greenleaf General • Halo • Panel • PC-lint • PforCe • Pre-C • Windows for C • Windows for Data C terp • db Vista • Phact • Plink86Plus • C-tree.

CP/M • TRS-80 • 8080/Z80 ROM

C compiler, 8080/Z80 assembler, linker, librarian, UNIX libraries, and specialized utilities.

Aztec C II-c (CP/M-80 & ROM). $349
Aztec CII-d (CP/M-80) $199
Aztec C80 (TRS-80 3&4) $199

Aztec C68k/Am 3.4

New Amiga Release

Amiga user groups across the USA voted Aztec C68k/Am release 3.3 the best Software Development System for the Amiga. Release 3.4 is more impressive.

Aztec C68k/Am-p Professional $199
A price/feature/performance miracle. System includes: optimized C • 68000/680x0 assembler • 68881 support • overlay linker • UNIX and Amiga libraries • examples.

Aztec C68k/Am-d Developer $299
The best of Manx, Amiga, and UNIX. System includes: all of Aztec C68k/Am-p • the Unix utilities make, diff, grep and vi.

Aztec C68k/Am-c Commercial $499
Aztec C68k/Am-d plus source for the libraries and one year of updates.

Aztec C68k/Mac 3.4

New Macintosh Release

For code quality, reliability, and solid professional features, Aztec C for the Macintosh is unbeatable. This new release includes features and functions not found in any other Macintosh C development system.

Aztec C68k/Mac-p Professional $199
• optimized C • 68000/680x0 assembler • 68881 support • overlay linker • UNIX and Macintosh libraries • examples.

Aztec C68k/Mac-d Developer. $299
The best of Manx, Macintosh, and UNIX. System includes: all of Aztec C68k/Am-p • the Unix utilities make, diff, grep • vi editor.

Aztec C68k/Mac-c Commercial $499
Aztec C68k/Am-d plus source for the libraries and one year of updates.

Aztec C65

New ProDOS Release

Aztec C65 is the only commercial quality C compiler for the Apple II. Aztec C65 includes C compiler, 6502/65C02 assembler, linker, library utility, UNIX libraries, special purpose libraries, shell development environment, and more. An impressive system.

Aztec C65-c Commercial $299
• runs under ProDOS • code for ProDOS or DOS 3.3

Aztec C65-d Developer. $199
• runs under DOS 3.3 • code for DOS 3.3

Aztec ROM Systems

6502/65C02 • 8080/Z80 • 8086/80x86 • 680x0

An IBM or Macintosh is not only a less expensive way to develop ROM code, it's better. Targets include the 6502/65C02, 8080/Z80, 8086/80x86, and 680x0.

Aztec C has an excellent reputation for producing compact high performance code. Our systems for under $1,000 outperform systems priced at over $10,000.

Initial Host Plus Target $750
Additional Targets $500
ROM Support Package $500

Vax, Sun, PDP-11 ROM HOSTS

Call for information on Vax, PDP-11, Sun and other host environments.

C' Prime

PC/MS-DOS • Macintosh
Apple II • TRS-80 • CP/M

These C development systems are unbeatable for the price. They are earlier versions of Aztec C that originally sold for as much as $500. Each system includes C compiler, assembler, linker, librarian, UNIX routines, and more. Special discounts are available for use as course material.

C' Prime $75

Aztec Cross Development Systems

Most Aztec C systems are available as cross development systems. Hosts include: PC/MS-DOS, Macintosh, CP/M, Vax, PDP-11, Sun, and others. Call for information and pricing.

How To Become An Aztec C User

To become a user call 800-221-0440. From NJ or international locations call 201-542-2121. Telex: 4995812 or FAX: 201-542-8386. C.O.D., VISA, MasterCard, American Express, wire (domestic and international), and terms are available. One and two day delivery available for all domestic and most international destinations.

Aztec C is available directly from Manx and from technically oriented computer and software stores. Aztec Systems bought directly from Manx have a 30 day satisfaction guarantee.

Most systems are upgradable by paying the difference in price plus $10. Site licenses, OEM, educational, and multiple copy discounts are available.

MANX

To order or for more information call today.

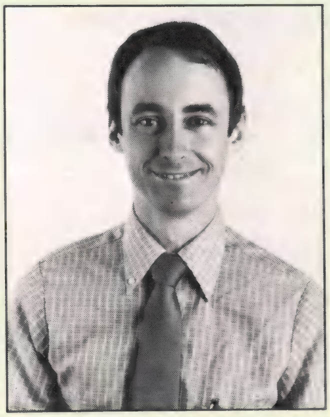
1-800-221-0440

In NJ or international call (201) 542-2121 • TELEX: 4995812

CIRCLE 108 ON READER SERVICE CARD

Manx Software Systems
1 Industrial Way, Eatontown, NJ 07724

RUNNING LIGHT



By way of introduction, let me say that the image on the right may look like the new editor of *DDJ*, but it isn't. Sure, we both share the improbable name of Tyler Sperry and we're both generally the same size and shape. But don't be fooled. Anyone who knows me will instantly recognize the fellow in the photograph as an imposter because he's wearing (horrors!) a tie. As a reformed hacker, I tend toward the view that t-shirt and jeans are adequate attire for nearly any social occasion.

Besides, I'm much better looking than he is. (Trust me.)

If you were to visit the real me at the *DDJ* offices, you'd probably be shocked by a few other conflicts between that picture and reality. In contrast to the benign, seamless background of that photo, I spend my days (and sometimes nights) in an office crammed with books, papers, computers, and software. In short, it's your typical programmer's nest combined with editorial clutter.

Because this would be your first visit with the new editor, we'd probably spend most of our time getting to know each other. I'd tell you some wildly funny (and completely unprintable) stories culled from over 15 years of playing with computers. Stories about my hardware days playing with the innards of video games or my time working in Kaypro's engineering department. There'd be some libelous stories from my stint as editor and publisher of *Profiles*, Kaypro's user magazine.

After all that talk about me, I'd break out a couple of cans of Jolt and we'd start talking about you and your interests. (I'm always on the lookout for new writers for *DDJ*.) We'd discuss the Intel and Motorola processors, and I might even confess some of my youthful indiscretions with the RCA 1802 and National's 16000 family.

DDJ being the kind of magazine it is, sooner or later we'd have a religious argument about languages. You'd probably start innocently enough with a comment about your favorite language, and I'd talk about how I've tried a dozen computer languages from ALGOL to Z80 assembly and never found one that satisfied all my desires. If that didn't work, I'd try provoking you by admitting that I do most of my programming in either Forth or assembly language. Eventually you'd rise to the bait and I'd have snookered you into writing an article that demonstrated the virtues of your favorite language in a way that none of our readers had ever seen before. A conniving lot, these *DDJ* editors.

Of course, you've only been visiting me metaphorically, so you'll have to take my word for how funny the stories are. And you'll also have to remind me by letter, CompuServe (76703,4266), or phone ([415] 366-3600) of what sort of article you were planning to write for us and if you need one of our writer's kits. The October issue will be all set by the time you read this, but there's probably still time to get an article into our November graphics issue. We're always looking for good articles, whether or not they reflect a particular issue's focus.

Now, if you'll excuse me, I have a magazine to get back to. Thanks for stopping by, and stay in touch.

Tyler Sperry

Tyler Sperry
editor

ARCHIVES

Windows

"Three examples: VisiCorp, Microsoft with Windows, Digital Research with Concurrent DOS. In all three cases we have consistent out-of-schedule behavior. These were basically all companies that went from very small development groups to significantly larger development groups, and probably without project management."—*Giacomo Marini, quoted in The Software Designer,* DDJ, *October 1984.*

Learning from Mistakes

"One of the major frustrations in my life is my inability to remember to use facts and ideas I know perfectly well. When I look at programs written by my friends, I suspect the affliction may be well-nigh universal."—*Programming Pastimes and Pleasures, Charles Wetherell,* DDJ, *February 1979.*

"Have you ever put two (or more) memory boards into the same address location? If they are in RAM, the computer will work fine, except that there will be a lack of memory. This program scans all addresses and displays a map of memory as seen by the processor. This 'lack of memory' will be caught quickly and hopefully save some time from head-scratching."—*"Memory Map Program for the Z80/8080," Robert Alkire,* DDJ, *January 1979.*

Ten Years Ago in *DDJ*

"In the early days of personal computing (i.e. last year) hobbyists were happy to be able to display anything, jerkily or otherwise. Today, with lots of video displays available, byters can afford to be choosy. One of the features I'm looking for in my next video display is linear scrolling, where the whole page moves up one raster line at a time. Has anybody seen one?"—*Jim Day,* DDJ, *June/July 1977.*

"A San Francisco dealer in used computer equipment announced at the July 20th Homebrew Computer Club meeting that he has dozens of properly functioning 2311-type hard-disc drives available for $350. Several people in both northern and southern California are currently completing controllers for these *7 1/4 megabyte* drives that will plug into a S-100 bus."—DDJ, *June/July 1977.*

"TRIPPY COMPANY NAME NOTED

"Joining the ranks of such straight-laced companies as Parasitic Engineering and Barefoot Computer Store, we recently noted a new California company, specializing in repairing sickly equipment: Micro-Mouse. Now, with a name like that, you *know* they gotta be good people."—DDJ, *June/July 1977.*

DR. DOBB'S JOURNAL of COMPUTER Calisthenics & Orthodontia
Running Light Without Overbyte

LETTERS

Artificial Neural Network Pioneers

Dear *DDJ*,

I read with considerable interest "An Artificial Neural Network Experiment" by Robert Brown (April 1987). This article is a classic example that nothing is really new; except for the modern implementation using state-of-the-art hardware, virtually everything described in this article was first published in the early 1960s.

In 1961 and 1962, Dr. Bernard Widrow of Stanford University first published the basic theory described in this article; he reduced it to practice in a device known in those days as Adaline and even formed a corporation to manufacture the adaptive elements used in the Adaline device. Adaline was implemented using essentially the circuit described in Figure 4 of the article and used electrochemical cells whose resistance could be modified by plating or stripping material from the cell plates. Thus, an adaptive network exactly as described in the article was achieved. The mathematics involved was identical to that detailed in the article.

Adaline was reduced to practice in a 4×4 array of cells and was demonstrated to show pattern recognition characteristics identical to those described in Table 1 of the article; an expanded Adaline (Madaline) was demonstrated to learn how to balance a broom on a model railroad car after being taught for a few minutes by an operator. The algorithm was implemented on a computer and demonstrated to have speech recognition characteristics that were remarkable for the state of the art existing at that time (1962).

Widrow is well published in this area—for example, see "Rate of Adaption in Control Systems," *American Rocket Society Journal* (September 1962): 1,378–1,385.

Subsequent to Dr. Widrow's work, a major extension of this work was published by Dr. John Voevodsky, which deals extensively with the neurological structure and its relationship to the work by Dr. Widrow.

I was a student of Dr. Widrow at the time the Adaline work was in process and have wondered over the years why nothing became of it. Much of the inhibition was caused by the cold water poured on this topic by Marvin Minsky in his book *Perceptions* in 1969; the lack of suitable technology to scale the concept up to useful levels was also a practical inhibitor.

Dr. Voevodsky has continued to champion the concept over the years and has recently formed a corporation to advance this technology (see *Electronic Engineering Times*, [March 2, 1987]: 24).

I applaud Robert Brown for presenting this information clearly in a modern context, but it is important and instructive to understand the true origins of this work and to give appropriate credit to Bernard Widrow and John Voevodsky for their truly pioneering work in this field.

David Lytle
90 Sidney Ct.
San Rafael, CA 94903

Educating Programmers

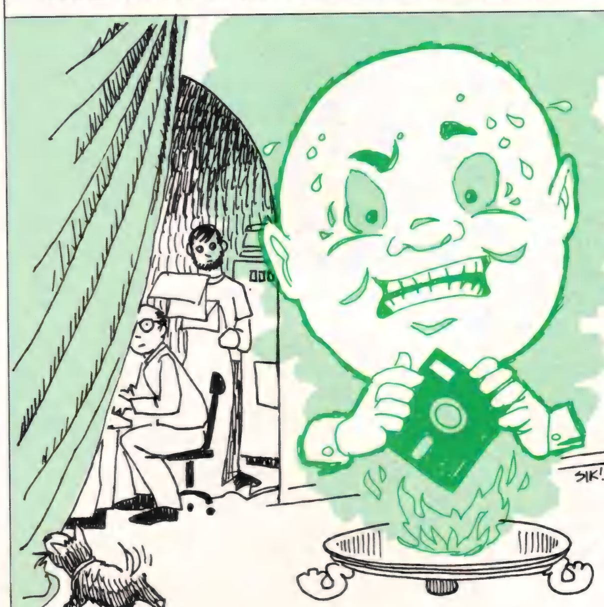
Dear *DDJ*,

I would like to make a few comments on Allen Holub's Viewpoint in April 1987.

He has a good point; however, how about going one step farther and taking issue with the very way math is taught in undergraduate classes? It's true that right now calculus means successive application of memorized rules until you find the right one. But that need not be. In fact, I suggest that it gives rise to poor math—not only poor programming. Undergraduate math (and science) education in the U.S. lags far behind the level of most other industrial countries, and that kind of syllabus—implying that you don't really have to apply logical thinking, understanding of the problem at hand, and so on—helps a lot toward this negative goal.

I agree completely that a good liberal arts curriculum is best for programmers—in fact it's best for scientists, too. Incidentally, less memorizing and more understanding of what's going on could make calculus (or any math, for that matter) an integral part of such a curriculum and lead to better programmers, better scientists, and, maybe, just plain better human beings (not to speak of English majors). By the way, that's what people try,

THE GREAT AND POWERFUL OS



"Pay no attention to those men behind the screen; Let's discuss my fabulous VAPORWARE . . ."

A Multiwindow Screen Editor for Programmers

XTC is more than just a powerful programmer's editor. Its implementation actually encourages users to develop software systems in a more efficient way.

XTC was originally written for a Compupro 8086 machine running CP/M-86, for use in robotic software development. It was called GLACIER, an acronym for "Good Lord, A Centralized Interactive Editing Routine". GLACIER had multiple windows and the ability to share text between windows, and a set of editing commands which were based on the most popular word processor at that time —Wordstar. Windows were important because it was necessary to view several different text files or several different parts of the same text file without constantly switching back and forth between them. Originally written in Pascal MT+86, GLACIER was rewritten for the IBM PC family in IBM Pascal. Blocks, text buffers, and a macro programming language were added to the new editor, and it was released as XTC version 1.2.

Now, nearly three years later, XTC 3.0 is a complete text editing environment with over 100 commands in seven categories: File Interface, Windows, Blocks, Text Processing, Editor Programming, Multitasking, and Special Editing. It comes with its source code on disk, and a full-size, laser-typeset, 180+ page manual. Unlike other programmer's editors, XTC is user-modifiable at the source code level, so you can add to XTC's basic command set, or change how the commands work entirely.

How XTC works

The editor's design was based on its ability to share text across windows and buffers (which are just windows that don't get displayed). Pascal's strong typing lead to the storage of text in a doubly-linked list of line buffers (hereafter called a "*text stream*"), so that cursor movement commands could move a "current pointer" around the text with simple pointer assignment operations. This also makes insertion of text very efficient, since it can be accomplished with one short memory move to make room for a new character.

XTC keeps track of a doubly-linked list of data structures called *window descriptors* which describe the windows displayed on the screen. There is also another doubly-linked list of window structures which are not displayed by the display algorithm for XTC's buffers. Each window descriptor contains information about the physical coordinates of the window on the screen, how big the window is, a pointer to a text stream where the window is positioned over, and a pointer to the current cursor position within the window. Other information, such as the current filename and options like wordwrap and autoindent are also saved in the window descriptor. Figure 1 shows how XTC's principal data structures are connected.

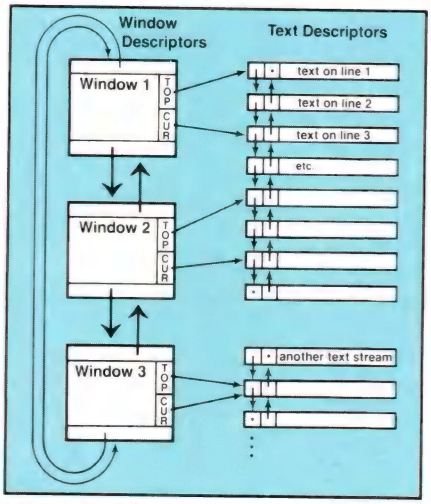


Figure 1. XTC's text and window management data structures. Here two windows share a text stream, and a third is positioned over a different stream.

How to use XTC

Knowing about how a program works makes it possible for you to use it in the best way possible. Obviously, XTC's data structures make for very fast zooming through a text stream, and insertion. It also makes it easy to make new windows quickly and either share text among them or look at other files while you concentrate on the main task of creative development.

There is no free lunch, unless you learn the system.

It takes XTC longer to read a file into data structures, and allocate memory for data structures on the fly for each line descriptor than it does for an unstructured editor to read text into big block buffers. Therefore, programs developed under XTC tend to be composed of a main file and many *include* files. XTC's main draw then, is its windowing system which makes you actually want to put a separate C function in a different *include* file. By storing each function in a separate file, XTC lets the operating system manage your library of source, and it becomes easy to share source routines among several software products.

This system works very well. We've used it in the design of our latest operating system, Wendin-DOS. The new operating system is composed of over 32 modules written in C and assembly language, and each module source file (named *modulename.C*) uses *include* statements to draw in header files that describe data structures, and function files (named *functionname.I*). Altogether, XTC and the operating system manage over 50,000 lines of source code in 500 different source files. The simple convention of using a function's name for its source file makes it easy to retrieve the exact piece of code you're interested in, without wading through tons of code in a single big source file.

Next month, we'll present XTC's programming commands and macro languages. If you don't already have XTC, get your copy and follow along. You simply won't get the chance to do that with any other editor.

XTC: $99
Wendin, Inc.
P.O. Box 3888
Spokane, WA 99220
(509) 624-8088

(continued from page 10)

with varying success, to do when teaching math in Europe.

Federico Marchetti
2505 Jemsen Ave., #438
Ames, IA 50010

Dear *DDJ*,
This is a response to the Viewpoint "Education and Programming" by Allen Holub in your April 1987 issue.

This essay touched on a potent mystery: exactly what is the knowledge or state of mind required for good programming? That is, what is programming? Hackers, have you ever been stymied by a persistent novice who keeps asking you "Yes, but what is it that you do?" Mr. Holub may approve of what I consider the best answer so far: "I write novels that computers read" (Patrick Hogan, 1987). I, too, believe that programming is, above all, a literary skill, an art of written communication. Look at the coincidences in the qualities of good writing and good programming: concise (efficient), consequential (functional), powerfully expressive (good use of hardware), evocative of hidden truths, structurally harmonious at scales both large and small.

That's why I think the notion that mathematics is inessential to programming is wrong. On the contrary, mathematics can teach things needed by writers of software, newspapers, and poetry alike. The first of these is logic, the foundation of mathematics. Impatient novelists and frobbozzers would not shudder if they could learn how quirky and "illogical" mathematical logic really can be. But it don't mean a thing if that proof don't sing! From logic you learn that the truth, to be valuable, must be communicated, that it must be notated, that it pays to use powerful theorems, and that the shortest road home is the best. And, as an inspiration, there is the mystical promise of Gödel's theorem—that there is something so true that it cannot be captured in proof.

For programmers, logic offers such especially apt vocational training that it deserves extra study and practice, long after the premeds and lit jocks have moved on to Latin. Here, programmers can acquire a handy facility with abstract symbols and an assertion-and-proof style of thought that, as Dijkstra and Mills have demonstrated, is practical and effective. Mathematics courses are good practice in logic and following "the drill." Computers and math quizzes are equally stern in their judgment, so take plenty of math and get used to it. Plus, along the way, you'll pick up plenty of useful tools and tricks. I pity graphics programmers who have not studied linear algebra and real analysis, for they are doomed to write inferior code.

If, as Mr. Holub suggests, you limit your study of mathematics to one semester of basics, you will not only have failed to acquire a good liberal arts education but you will also have fallen behind in the art of programming.

Kerry Kimbrough
1710 Aggie Ln.
Austin, TX 78757

Dear *DDJ*,
Allen Holub fails to see the very deep connection between mathematics and programming. The facts of the matter are that mathematics and programming are inextricably related. Both share a common ground, namely that of solving problems. But before a solution to a given problem is achieved, the nature of the problem must be understood—that is, the specifications of the problem are given. It is here that mathematics and programming have the greatest similarity. Both methods produce their solutions from what is known about the problem in advance, and the more that is known, the easier the solution. Generation of the solution is performed in a logical and orderly fashion. Solution of math problems is not haphazard, despite Mr. Holub's statement to the contrary. In fact, many of the items in his assessment of essay-writing fundamentals are also found in mathematics. The mathematical analogue of the outline are the axioms, or those things given as true, of the problem. Also, solutions to mathematics problems are generated a step at a time, similar to the sections of an essay. In either case, methodical and ordered processes are performed in order to achieve the desired result.

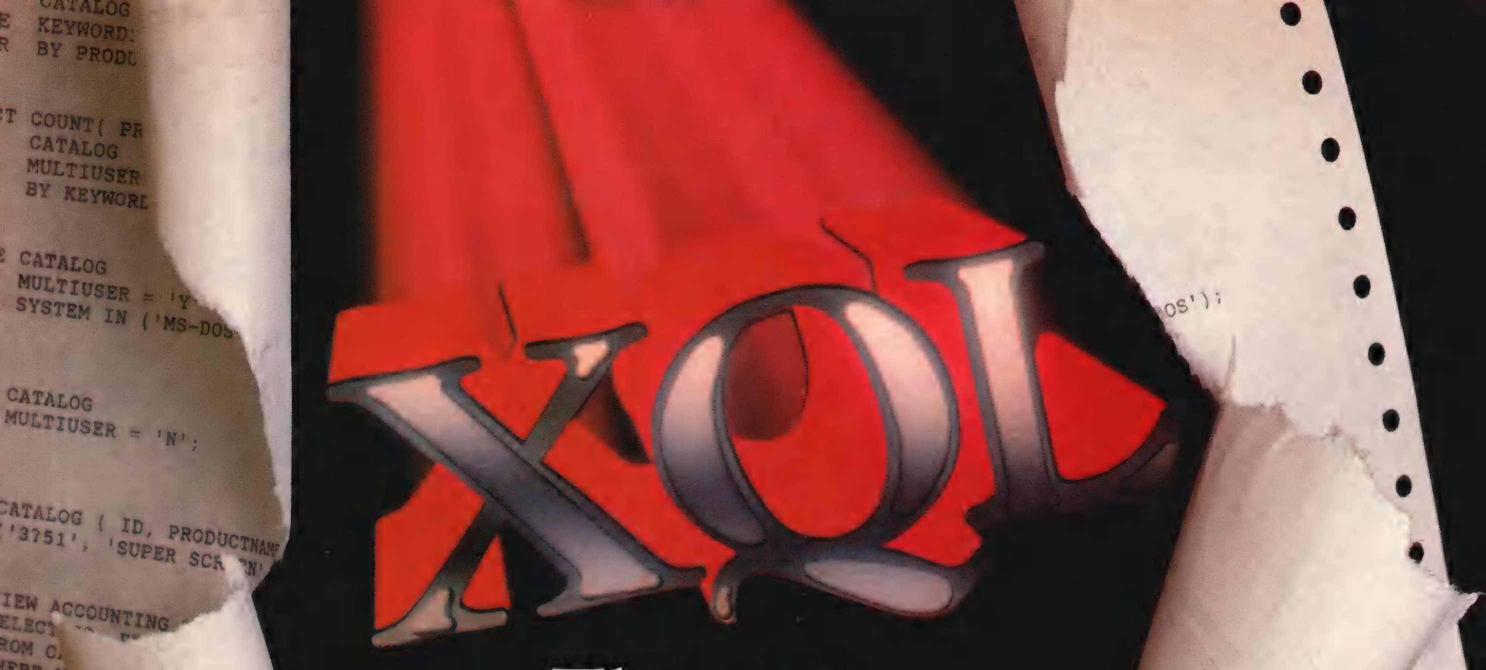
Mr. Holub's suggestion to use Latin as the paradigm of complex systems, though laudable, is flawed. Latin, although rich in its complexity, is a dead language: its patterns are fixed and unchanging. Computer programs, on the contrary, are dynamic, evolving entities. Latin is, therefore, inadequate as a model from which to study the structure of programs. Human language is also fraught with inconsistencies—for example, exceptions to rules—and sometimes suffers from ambiguities and imprecision. Programming requires quite the opposite, as even novice programmers could testify. Mathematics is also intolerant of ambiguity and imprecision. Through the use of well-defined, consistent principles, solutions to mathematics problems are guaranteed to be true to a high degree of reliability. No human language can assert the same.

Mathematics takes several years to grasp fully. A one-year calculus class is sufficient to provide only a coarse introduction to mathematics. Rather than see mathematics requirements be watered down, curricula should be expanded to include upper-division mathematics requirements.

One final thought in this area: training in language and literature, although providing a broad knowledge base, does little to provide students with the necessary problem solving skills and methodology applicable to computer programming. What problems are there to be solved in Russian literature?

We do not need groups of people capable of writing text editors, database managers, and so on. Surely there are enough of these programs already. "Scientific" programming should be strongly emphasized in any programming curriculum. Mr. Holub is correct in his assessment that certain aspects of undergraduate education require reform, but he has failed to see the deeper underlying problem: the need to make mathematics, science, and computers, because of their increasing importance, more available to undergraduates, even those who, in Mr. Holub's words ". . . don't have an aptitude for

(continued on page 126)



The most brilliant breakthrough in SQL technology since SQL.

XQL is a dramatic step forward in the history of SQL. It's the one unique SQL solution that helps programmers break through to even higher levels of productivity. Powerful yet easy to use, XQL minimizes your coding time and lets you focus on building better applications.

XQL extends the power of Btrieve, SoftCraft's high-performance file manager, by allowing access to multiple records at a time. It frees your application from physical file characteristics by providing true relational capabilities with data independence, data descriptions, data integrity and security.

XQL's three interface levels are a major advance in SQL technology. The first two levels, XQL primitives for maximum efficiency or full SQL statements for maximum convenience, are callable subroutines from BASIC, Pascal and C. The third level lets you enter SQL statements interactively without ever having to write a program.

XQL's extensive DBMS features let you access data by name. Field order is independent of physical location within the Btrieve record. Only records that pass your restrictions are returned—in the sort order you specify. Fields can be computed from other fields or constants. And you can manipulate composite records built from multiple, joined Btrieve files.

XQL offers all the performance and reliability you've come to expect from Btrieve, including LAN support, fault tolerance, comprehensive documentation and expert technical support for trouble-free software development. Plus, you never pay royalties on your XQL applications.

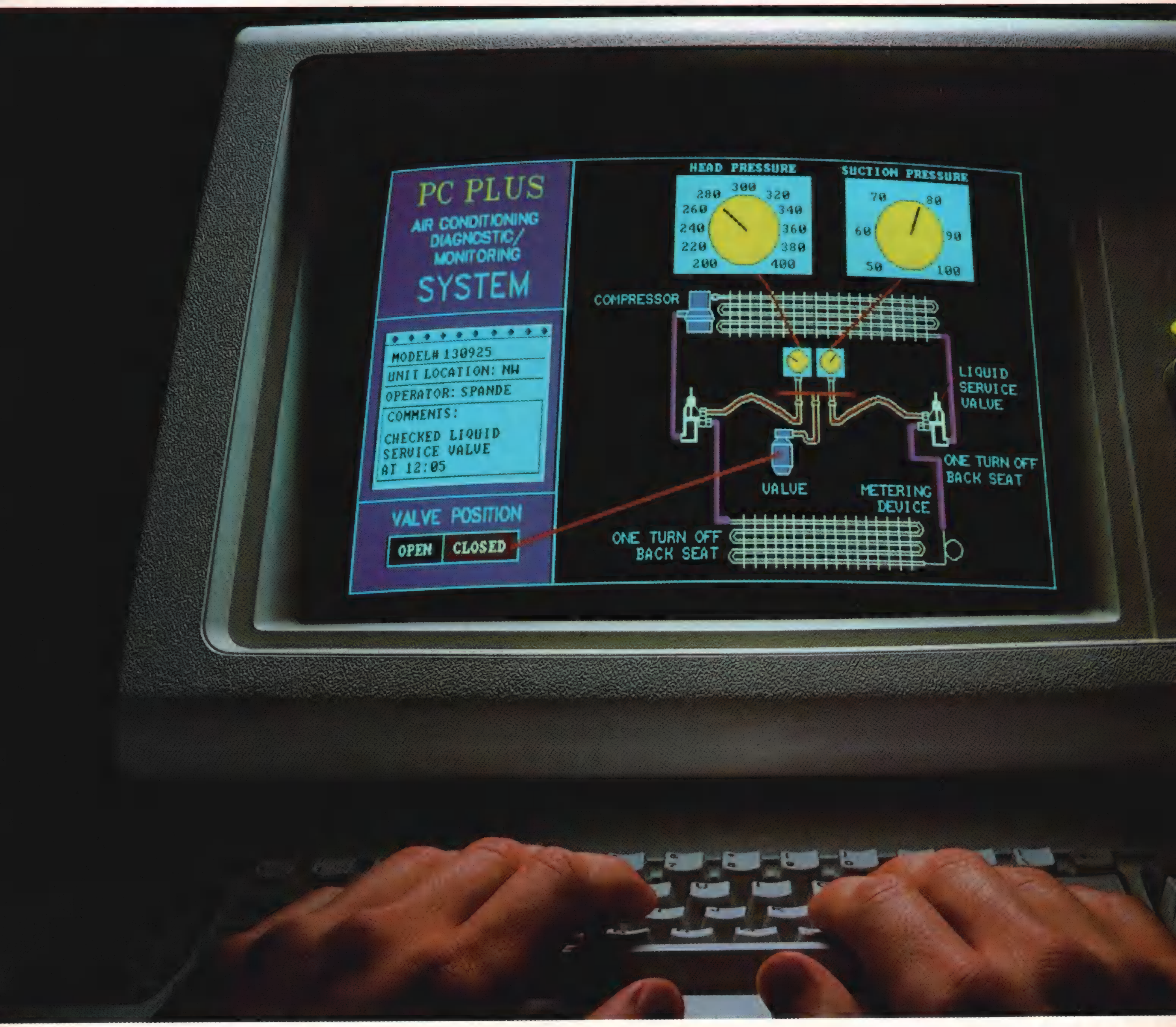
Put the latest innovation in SQL technology to work for you. Contact SoftCraft.

SoftCraft
A NOVELL COMPANY
P.O. Box 9802, #917
Austin, Texas 78766
(512) 346-8380 Telex 358 200

XQL, $595; Btrieve, $245; multiuser Btrieve, $595. XQL requires Btrieve and PC-DOS or MS-DOS 2.X or 3.X. XQL is a trademark and Btrieve is a registered trademark of SoftCraft, Inc.

CIRCLE 113 ON READER SERVICE CARD

Texas Instruments has system developers need.



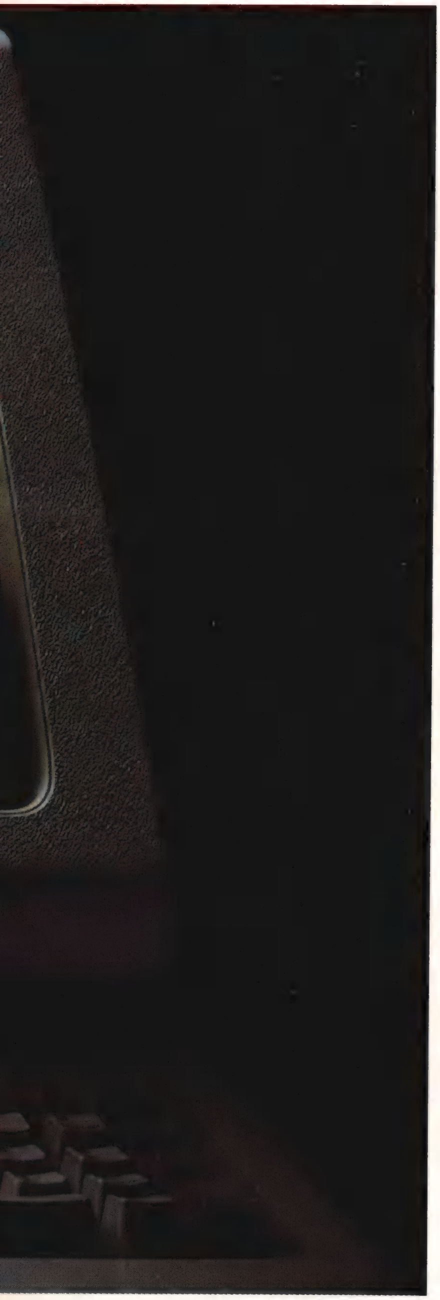
"Personal Consultant™ Plus . . . offers a very fine expert system development and delivery tool that already has a proven record with end-users."

— *Susan Shepard, AI Expert*

Personal Consultant Plus 3.0 Standard Features
- Frames, rules, meta rules and procedures
- Forward/backward chaining
- Confidence factors
- Regression testing and rule tracing
- End-user explanation facilities
- Graphics image capture and display
- Interfaces to dBase™, Lotus 1-2-3™, DOS files, .EXE or .COM programs, "C"
- Complete LISP development environment
- 2-megabyte expanded/extended memory support
- Mouse support
- Context sensitive help
- "Getting Started" tutorial-style manual

Personal Consultant Images
- Optional add-on package to PC Plus (3.0)
- Allows integration of "active images" into

what serious expert Power tools.

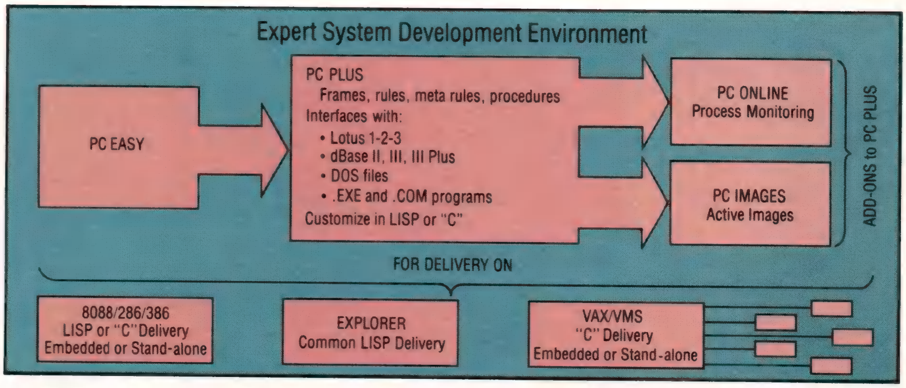


knowledge bases
- Interactive dials, gauges, forms and selection images
- Multiple images can be combined on same screen
- "Getting Started" tutorial-style manual

Personal Consultant Online
- Optional add-on package for PC Plus (3.0)
- ONLINE expert systems that interact directly with process data
- Multiple interfaces to data acquisition and analysis programs
- Knowledge base synchronization with process data
- Functions for historical and predicted trends
- Special user interface/reporting capabilities
- "Getting Started" tutorial-style manual

Among all the expert system development tools available for personal computers today, none deliver the power and flexibility of TI's Personal Consultant series.

Personal Consultant Easy is ideal for getting started, and is upwardly compatible with the higher functionality of PC Plus. For experienced developers, Personal Consultant Plus and its optional add-on enhancements, Online and Images, were designed to help solve a broader range of complex problems.



Personal Consultant Plus. Full power for an affordable price.
At $2,950, PC Plus has proven to be one of the richest and most flexible problem-solving tools available for the development of complex knowledge-based systems. Designed to take advantage of today's more powerful 286/386 DOS-based computers, or TI's Explorer™ Symbolic Processing System, the new 3.0 version of PC Plus provides powerful standard features and a continuing growth path with the addition of either PC Images or PC Online, or both.

Personal Consultant Images. Picture an expert system with interactive graphics.
At $495, PC Images enables developers to create knowledge-based applications that incorporate complex graphical "active images." User-interactive dials, gauges, forms and selection images provide a more exciting visual data input and output style.

Personal Consultant Online. The expert system as part of the process.
At $995, PC Online allows the developer to design expert systems which interact directly with process data, as opposed to input from a human operator. Designed for intelligent process monitoring applications, this optional package helps deliver expertise that is "online all the time."

Application delivery as flexible as the tools themselves.
Delivery can be in LISP for flexibility, or "C"* for maximum speed and portability. Our "C" options support either stand-alone or "embedded" knowledge bases. Options are available for DOS-based PCs, TI's Explorer, and DEC's VAX™ line of multi-user minis running under VMS™.

"Texas Instruments has done more than any other company to educate people about AI, to popularize it, and to make useful AI tools available at reasonable prices."
— *Jim Seymour, PC Magazine.*

Technical support, training courses and Knowledge Engineering Services are available for the Personal Consultant products. If you have a question about any of our expert system power tools, we have the answer.

Pick up the phone and gain a powerful advantage.
Call **1-800-527-3500** for technical overviews of our products and a PC Plus case histories brochure which details how our power tools are being put to work today.

TEXAS INSTRUMENTS

Developing 80386 Applications . . . Today

by Richard Relph

There are alternatives to waiting a year or two for Microsoft's 386 version of OS/2.

The arrival of the Intel 80386 promises a new level of sophistication in applications. With its speed and multigigabyte virtual, physical, and segment address spaces, developers can create applications requiring the resources of a mainframe but usable on personal computers. In addition to being able to run these new, sophisticated applications, the 386 can play host to several "old" 8086 programs, each believing it has the whole CPU to itself. It is this promise of being able to run newer, more powerful applications without losing the use of existing programs that makes the 386 so exciting.

But many software developers may feel the availability of a new CPU such as the 386 is difficult to take advantage of. There isn't an operating system supporting the 386 that people are willing to buy. And even when Microsoft's 386 DOS becomes real, the Lotuses of the software world will get first crack at it, taking away the small developer's edge of being able to react more quickly. It seems as if there's no way for a small company or individual to develop the first spreadsheet, for example, to take advantage of the 386's capabilities.

Well, that isn't actually the case anymore. Just about any programmer can develop code today that takes advantage of the full resources of the 386, using what I call an "environment." An environment is a layer of software that looks like a 386 DOS to your 386 application but that acts like a standard 8086 application to the host operating system, MS-DOS. This article describes two such environments that were available in March; at least two more should be available by the time you read this.

The environments described here limit your 386 application to physical memory and must be written in either Pascal or C (unless you feel up to developing multimegabyte programs in assembly language). Over the next few months, these restrictions may be eased. Environments supporting virtual memory and other languages will certainly be ported to the 386. Two environments supporting multitasking were planned for release in April.

Before you can use an environment, you need a compiler that supports the environment (and its host processor, the 386). First, you select your compiler; then an environment; and finally, the hardware to run it on. All three ingredients are available today.

The only vendor supplying 386 compilers for any environment I know about is MetaWare, which is now shipping both Professional Pascal and High-C. Phar Lap was the first company to ship an environment—DOS-Extender. Softguard followed soon after with VM/RUN, the pieces of VM/386 needed to run 386 programs. The Software Link provides PC/MOS 386, a multiuser, multitasking OS for the 386. The version of PC/MOS I tested supported only 8086 tasks, although the company provided documentation and release dates for the 386-tasking version. The hardware I used was the Compaq Deskpro 386.

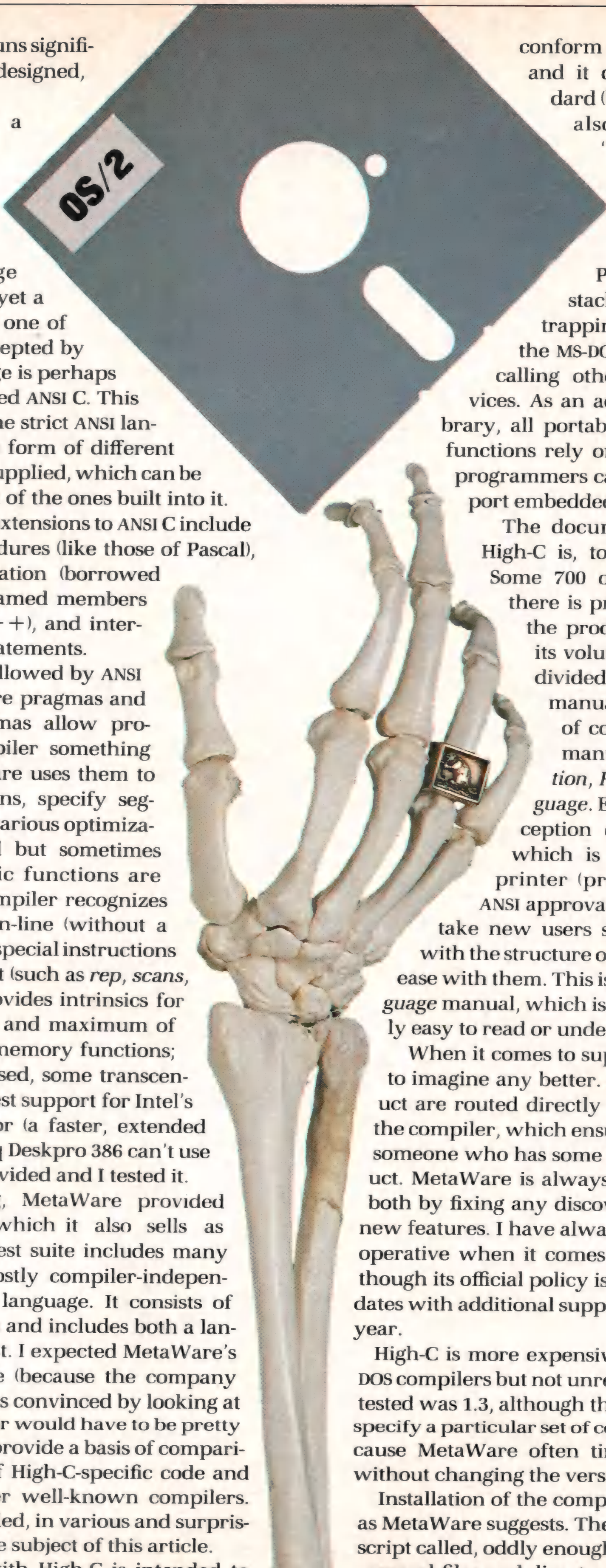
The Compiler

Compiling your source code is the starting point for creating 386 programs. I used MetaWare's High-C 386 to perform this task. High-C 386 itself runs on any MS-DOS computer with a hard disk. In fact, no 386 is required anywhere in the development process until you actually need to execute (and debug) your code. Of course, like any

Richard Relph, 846 Salt Lake Dr., San Jose, CA 95133. Richard is a software and hardware consultant. He has written compilers and embedded systems.

MS-DOS program, High-C runs significantly faster on a well-designed, 386-based computer.

MetaWare High-C is a complete (and then some) implementation of the C language, regardless of which standard you choose to hold it up to. The ANSI C language specification, though not yet a standard, is supported in one of two similar languages accepted by High-C. The other language is perhaps best referred to as extended ANSI C. This is the default language. The strict ANSI language is supported in the form of different parse and scan table files supplied, which can be fed to the compiler in lieu of the ones built into it.



Some of the MetaWare extensions to ANSI C include case ranges, nested procedures (like those of Pascal), named parameter association (borrowed from Ada), access to unnamed members of unions (taken from C++), and interleaved declarations and statements.

Two unusual features allowed by ANSI and used by MetaWare are pragmas and intrinsic functions. Pragmas allow programmers to tell a compiler something about their code. MetaWare uses them to change calling conventions, specify segments for objects, enable various optimizations, and other unusual but sometimes needed features. Intrinsic functions are function calls that the compiler recognizes and generates code for in-line (without a procedure call), using any special instructions the processor may support (such as *rep*, *scans*, *movs*, etc.). MetaWare provides intrinsics for absolute value; minimum and maximum of lists; common string and memory functions; and, if the 387 option is used, some transcendental functions. I didn't test support for Intel's 80387 numeric coprocessor (a faster, extended 80287) because the Compaq Deskpro 386 can't use one, but 287 support is provided and I tested it.

To do language testing, MetaWare provided me with its test suite, which it also sells as a separate product. The test suite includes many strange but valid and mostly compiler-independent constructs in the C language. It consists of roughly 2,000 lines of code and includes both a language test and a library test. I expected MetaWare's compiler to pass the suite (because the company provided it) and it did. I was convinced by looking at the test suite that a compiler would have to be pretty sound to run it, but just to provide a basis of comparison, I stripped the suite of High-C-specific code and ran it through some other well-known compilers. The other compilers all failed, in various and surprising ways, but that is not the subject of this article.

The library provided with High-C is intended to conform to the ANSI specification, and it does. Additional, nonstandard (but common) functions are also supported via utility "packages." These packages are in the form of header files much like the normal ones, only with the extension .CF. Packages are provided for stack dumping, for interrupt trapping (yes, trapping), for using the MS-DOS *int 21* functions, and for calling other system-dependent services. As an additional feature of the library, all portable, but system-dependent, functions rely on a core of functions that programmers can replace in order to support embedded or hostless applications.

The documentation provided with High-C is, to say the least, complete. Some 700 or more pages in length, there is probably no question about the product not answered within its volume. The documentation is divided into five sections, each a manual itself, each with a table of contents and an index. The manuals are *License*, *Installation*, *Program*, *Library*, and *Language*. Each is typeset, with the exception of the *Language* manual, which is printed on a dot-matrix printer (presumably awaiting final ANSI approval of the C standard). It will take new users some time to get familiar with the structure of the manuals and to feel at ease with them. This is especially true of the *Language* manual, which is precise but not particularly easy to read or understand.

When it comes to support for High-C, it is hard to imagine any better. Questions about the product are routed directly to the person who wrote the compiler, which ensures that you are talking to someone who has some familiarity with the product. MetaWare is always improving its compilers, both by fixing any discovered bugs and by adding new features. I have always found the company cooperative when it comes to providing updates, although its official policy is three months of free updates with additional support at 15 percent ($135) per year.

High-C is more expensive than MetaWare's other DOS compilers but not unreasonably so. The version I tested was 1.3, although this is not enough to exactly specify a particular set of compiler characteristics because MetaWare often tinkers with the compiler without changing the version number.

Installation of the compiler is easy if you proceed as MetaWare suggests. The first step is to run a batch script called, oddly enough, *install*. This script places several files and directories in the current directo-

(continued from page 17)

ry. It is unfortunate that most of these files are packed in a single archive file spread across the four diskettes. This makes customized installations a two-step process: first installing per MetaWare guidelines, then rearranging for individual tastes. To MetaWare's credit, the entire installation section of the manual is duplicated on the first diskette for easy reference.

Besides the compiler, include files, and necessary libraries, the package contains several utilities, such as some standard Unix-style file manipulators. More unusual is a set of utilities that allows editing and detailed examination of .OBJ files and utilities for producing cross-references of multiple source files.

Performance

The compiler isn't going to win any contests for speed of compilation, but users should find performance adequate for most programs. Most of the slowness comes from the size of the compiler and the time it takes to load from disk, so performance will appear particularly bad with small source files. The compiler does generate a lot of information about your program and issues many useful warnings when it finds questionable code, such as using a variable before assigning anything to it or failing to specify a return value from a nonvoid function. Listings, with or without generated assembly-language code, can be produced via a command-line switch.

The generated code is good considering that no global optimizations are made. Of course, the 386's more regular architecture doesn't hurt either, although there are nuances there as well. For instance, the fastest way to multiply something by 5 is to use the *LEA* instruction, of all things! High-C supports three register variables, and they are used by default, unlike High-C for the 8086 and 80286, which only supports two and then only if you enable them. The size of integers is 32 bits in High-C. I didn't spend a lot of time benchmarking the compiler because it is the only game in town right now. But I did compare programs compiled under 386 mode with programs compiled under 286 mode and found general improvements of 5 to 10 percent in the 386 version. In programs in which long arithmetic was used often, more dramatic increases were apparent.

The Linker

After compiling your program, you'll need to link it. I used Phar Lap's assembler and linker, although you can get the same assembler and linker from Softguard (which licenses them from Phar Lap). I did not play with the assembler much beyond running the installation test, but based on comments in the update notice, it seems fairly sound. The linker, on the other hand, I used a great deal.

386LINK, as it is called, is fairly slow, mostly because of the way in which libraries are managed. Unlike Microsoft's DOS linker, which uses an (undocumented) index table, Phar Lap's simply searches the library by scanning it. Doing all the arithmetic in 32 bits using 8086 instructions doesn't help performance either. The input form is something Phar Lap calls Easy OMF-386, a simple extension to OMF-86, the .OBJ form both Intel and Microsoft use for 8086 objects. Easy OMF-386 extensions are documented in an appendix of the linker's manual. The linker can produce several output forms. .EXE files are the default, although the format is not exactly compatible with DOS .EXE files because it doesn't use 8086-style segments. Intel hex and Motorola S-records can also be generated. A special output form is .REX, which is what the Softguard environment uses.

The Environments

If you buy the assembler and linker from Phar Lap, you also get MINIBUG and RUN386. The total package lets you develop programs to run on the 386. If you want to sell those programs, you must purchase a redistribution license. You can then bind RUN386 to your program and sell it. In this bound form, 386 programs are invoked just as normal DOS applications.

MINIBUG is a debugger for 386 programs, similar to MS-DOS DEBUG. Missing are the Assemble command, the Load command, and the Name command. The Dump and Enter commands are enhanced to support SYMDEB-style size specifiers for ASCII, word, and double-word quantities. The *RX* command displays all 80386 registers, while the *R* command displays just the registers used by normal programs. You can modify any 80386 register, including all the protected registers (which include the debug registers).

RUN386 is Phar Lap's 386 execution vehicle. It is a single, standard MS-DOS program that accepts the name of a 386 .EXE file, loads it, and runs it, passing any additional command-line parameters on to the 80386 application. RUN386's job has really only begun when your 386 program gets control, however. Presumably, your program will want to do I/O and probably through the MS-DOS that was running just before. The problem is that MS-DOS is an 8086 program and yours is not. Furthermore, you can address gobs of memory, but MS-DOS and its underlying hardware can only get at the first megabyte of physical memory. RUN386 handles the details of getting your data through to DOS and letting DOS get to its hardware. It does this by intercepting your *int 21*s, examining the registers, translating them and moving data if necessary, and finally passing control to MS-DOS in real mode. When MS-DOS finishes, RUN386 regains control and again translates registers accordingly. RUN386 also fields hardware interrupts and forwards them to their real mode handlers.

Softguard's VM/RUN does pretty much the same thing as RUN386 does. VMRUN is a .COM file that loads many other Softguard-supplied files (all of which must be in the current directory) and then the application's .REX file. Each application must have a profile, a description of how the programmer would like certain features of the environment configured. Mainly, these parameters specify how much memory to allocate for specific uses. Memory can be allocated for program-managed, low physical address buffers (thus avoiding the expensive block move), for DOS *exec* calls, and for the stack. All other memory is given over to the 80386 application code and its data.

Another bit of information in the profile is whether to start up in debug mode or not and whether to debug using the display or a separate terminal connected to a

(continued from page 18)

COM: port.

Softguard's debugger is different from Phar Lap's and is in many ways better. It uses the 80386's debug registers to set execution and data breakpoints. It is screen-oriented when running locally and can use a remote terminal as the debug console (in which case it does start to resemble DEBUG). When running locally, screen swapping is used to avoid the normal problems associated with one screen being used for two purposes. The debugger is more or less always present. If a trap occurs during execution of your program, the debugger is invoked.

One major difference between the environments these two packages present is the memory model. Phar Lap's continues to use segments, whereas Softguard's is based on a flat-memory model. Although it is accurate to say that Phar Lap supports the large model compared to Softguard's small model, it is perhaps a bit misleading. The small model, after all, supports up to 4 gigabytes per object or program. One place this dichotomy of models is apparent is in direct screen I/O (I do not recommend doing direct screen I/O, but both environments support it and it does provide a useful example of the difference between flat and segmented models). For Softguard's, you merely compute what the address would have been on the IBM PC (which depends on what display adapter is in use), put that in an index register, and access screen memory through it. So Softguard would have the screen addressed with *EDX* (for example) equal to 000b8000. Phar Lap, on the other hand, provides a segment descriptor that points at the base of the display adapter, so merely computing the offset and accessing through the segment with the offset does the equivalent thing. Segment 1c points at the screen memory, so to access the first location in it, you load a segment selector (register) with 1c and use an offset of 0. Phar Lap's 1c:0 is equivalent to Softguard's 000b8000, assuming a color graphics adapter. If you use a monochrome adapter, Phar Lap's environment still accesses it using 1c:0, but Softguard's uses 000b0000. It should be noted that Phar Lap does provide a segment selector (34) that works the way Softguard's *DS* does for the low 1 megabyte.

Softguard's VM/RUN is compatible with its future VM/386, a multitasking 386 control program. VM/386 is not an operating system but rather a layer between other operating systems and the hardware (VM/370 programmers should recognize this picture immediately). Under (or over, depending on how you view things) VM/386, several different 8086 operating systems may be running, each believing it owns the machine. One virtual machine can even be rebooted without affecting any other virtual machine.

The Software Link has taken a different tack. Its PC/MOS 386 is a single 80386 operating system that can run several DOS programs at one time. PC/MOS will also be able to support 80386 native mode programs, but this feature was not available at the time I wrote this review (it was due to be released in April). Documentation provided by The Software Link indicates that PC/MOS will support large-model 386 programs, like RUN386 does, although it doesn't appear that they'll be compatible.

Vendors

High-C 386
MetaWare Inc.
903 Pacific Ave., Ste. 201
Santa Cruz, CA 95060
(408) 429-6382
$895
Reader Service No. 40

386ASM, 386LINK, RUN386, MINIBUG
Phar Lap Software Inc.
60 Averdeen Ave.
Cambridge, MA 02138
(617) 661-1510
$495
$995, redistribution license
Reader Service No. 41

VM/RUN
Softguard Systems Inc.
2840 San Thomas Expressway, Ste. 201
Santa Clara, CA 95051
(408) 970-9240
$595
Reader Service No. 42

PC/MOS 386
The Software Link, Inc.
8601 Dunwoody Pl., Ste. 632
Atlanta, GA 30338
(404) 998-0700
$595, single-user
$995, multiuser
Reader Service No. 43

Performance

When comparing RUN386 and VM/RUN in use, RUN386 seems to be easier to use. Only one new file is introduced (RUN386.EXE) and it can be anywhere along the path, although the 386 .EXE file must be in the current directory. VM/RUN requires at least five other files, four of which must be in the current directory, along with the 386 application's .REX file. VM/RUN takes longer to load these files and to get to the task at hand (running your program) than does RUN386. RUN386 seemed to cooperate with my editor (Epsilon), whereas VM/RUN did not. Both VM/RUN and RUN386 did run under my make utility (Polymake). VM/RUN also clears the screen during initialization, which did not seem appropriate. And finally, VM/RUN is between 7 and 15 percent slower executing identical code on the same Compaq Deskpro 386. This is presumably because of hardware interrupt handling or overhead associated with running at CPL 3 rather than CPL 0 as RUN386 does. (CPL is the current processor level and represents the level of privileges that should be granted to a pro-

80386 APPLICATIONS TODAY
(continued from page 20)

gram—0 is the most privileged level.) Other explanations are possible, and the results may not repeat on non-Compaq 386 machines.

Softguard's product does outperform Phar Lap's in one important area—memory allocation. In addition to being more flexible (through the profile discussed above), Softguard's also seems to end up with more memory available. Running a binary search between 0 and *MAXINT* (2 billion), Softguard's reported 1,656,288 bytes available in a 2-megabyte system, whereas Phar Lap's reported only 1,496,912 bytes available. Of course, the more memory you have in the machine, the more memory will be available for 386 programs (until you hit that 4-gigabyte limit).

The Hardware

None of this discussion would matter if there were not computers on which to run these new environments. The Compaq Deskpro 386 proved to be an excellent performer on all counts. It was a joy to use. Everything about this machine is fast, except the tape drive. The machine has four speed modes—common, fast, high, and auto. Common speed is 4 MHz and is comparable to a 6-MHz IBM PC/AT. Fast is 8 MHz and is a bit faster than an 8-MHz AT. High is 16 MHz and is unlike anything you have had on your desk before. Auto mode switches between high and fast, depending on the diskette motor-on signal, attempting to ease speed incompatibilities in such areas as floppy-disk access. I ran in high-speed mode exclusively and had no problems, although I did not run any copy-protected software. The bus version of Microsoft Mouse also worked well.

Unless you believe the rumors about Intel building a 386 with 286 pinouts (shaving address bus and data bus pins in the process), I believe your next MS-DOS machine should have a 386 CPU. I know mine will. And more software developers have announced 386-specific applications in the six months since Compaq introduced its Deskpro 386 than all the 286 specific applications announced since, well, ever.

Summary

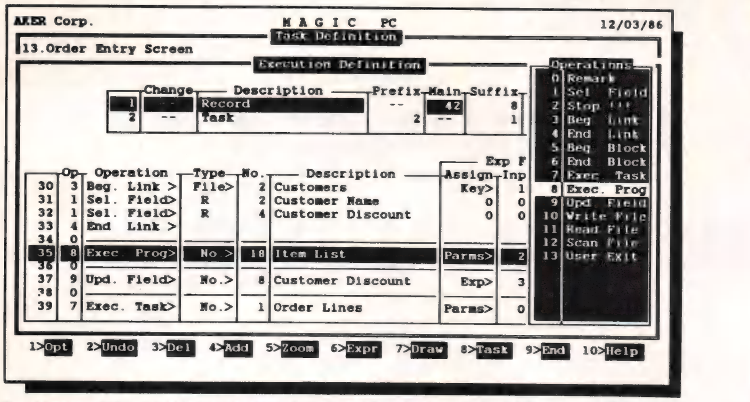
The point of this article is to convince you that you can get started developing 386-based applications now, without waiting for a 386 OS/2 to arrive, whenever that may be. If you develop software in a high-level language, such as MetaWare's High-C or Professional Pascal, you will at most have to recompile and relink your application to get it to run under some new environment.

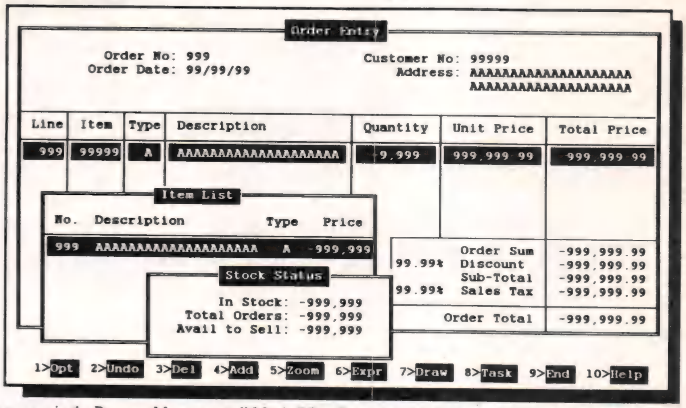
The tools to do these things are still young but are certainly adequate for conventional application development. They will undoubtedly get better as time goes on and probably are already significantly better than the tools I used to write this article in February and early March. I hope that anyone who wishes to do development for the 386 will contact each of the vendors mentioned here to get more up-to-date information.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

Develop DB applications 10 times faster without the coding pain...you'll swear it's Magic





Visual Programming with Magic PC looks as simple as this. To design an application you quickly place your design specifications into menu-driven Task Tables without having to write a single line of code. For example, just by highlighting the Execute Program operation on the left screen and also highlighting the Item List program in the Program Menu, you tell Magic PC to Zoom into the Item List program through the window shown on the right screen. The window will automatically scroll the Item List data horizontally and vertically, and allow query, "cut and paste" copy or even creation of new Items.

Free yourself from coding

Database professionals throughout the world are discovering a new way to dramatically cut development time.

So can you! With **Magic PC, the Visual Database Language** by Aker.

Consultants, VAR's, Software Houses and DP MIS professionals: If you develop DB applications for a living, now you can tackle any database application 10 times faster than with your DBMS or 4GL.

What makes you so fast with Magic PC?

It's not magic...it's simply because Magic PC finally frees you from coding. And doesn't coding take up most of your time right now?

Magic PC lets you leverage your design skills instead of wasting your time coding. Now you can generate a fully functional prototype in just hours for quick customer feedback, and easily refine the same prototype to a finished application.

All you do is enter your system design specifications directly into Magic PC's non-procedural menu-driven Tables, as ideas come to mind, and Magic PC generates the programs for you automatically.

Magic PC gives you a free hand to design powerful data management systems limited only by your own imagination. Without the time consuming mechanical details of conventional procedural programming. There's your competitive edge. The rest is up to you.

Your biggest time saving comes from Magic PC's dynamic adaptation to spontaneous design changes. You're free to change your design on the fly, and Magic PC automatically updates your programs and data files online. No more time wasted maintaining each program manually with every small change.

Visual Programming Power

You program with Magic PC by describing your data elements with **Data Dictionary Tables** (Files, Fields, Keys), and placing your system design specifications into **Task Description Tables.**

The Tasks can be nested within one another or dynamically Link to satellite Tasks, to give you true One-to-Many relational database power.

Only 13 Task **Operations** harness the power of Magic PC. Operations are specific enough to eliminate the need for tiresome coding, yet elastic enough to produce robust custom applications.

Use the Task building blocks to quickly generate Online Programs: Screens, Window Zooms, Menus; or Batch Programs: Reports, Updates, Data Import/Export and much more.

You develop the Task Tables visually on the screen by highlighting selections from Window Zooms and pop-up menu-driven Tables. You're not forced to follow any particular Table sequence, and there's no coding to slow you down. It's that simple.

You can apply mathematical and logical Expressions, or use the built-in Functions directly in the Task Tables to automate conditional Task processing, to display custom error messages or even invoke external applications such as spreadsheet, word processing or communication programs, transparently from within your Magic PC application programs.

Magic PC generates your application by fusing all your Data Dictionary and Program Tables seamlessly into a single **Integrated Library,** and automatically maintains changes online for optimal, bug-free performance, so you always get it right the first time.

Your application is executed at runtime by a **Magic Run** engine for stand-alone operation, and you can distribute your applications at a low cost and protect your design. Magic Run has a built-in visual interface to manipulate data and get on-the-spot ad-hoc information without any commands or syntax, simply by highlighting selections from menus. Data validation, security and error-checking are done automatically for you without programming.

Magic PC has built-in support for File and Record Locking so you can design multi-user applications for a local area network, and share data with any number of Magic Run users.

Magic PC integrates the Btrieve file manager internally, supporting the B-Tree file structure for fast high performance data access, and fault tolerant recovery during power failures.

Magic PC's powerful **Window Zoom** lets you design composite screens with windows to probe deep into the application through nested windows and manipulate the data underneath. By Zooming from any field, your end-user can conveniently query, copy or even create data in other programs directly through the windows, without stopping their screen session. The window frame size does not limit the available Data View since each window has built-in horizontal and vertical scrolling.

Magic PC is the professional's choice:

IBM France: "IBM encourages Magic PC and salutes such evolution..."

Israeli Air Force: "We were convinced that it was not possible to have a design tool powerful enough to implement real life applications without a programming language. Magic PC changed our mind..."

PC Magazine: "If the thought of programming database applications makes you tremble, Magic PC is for you. The applications generator saves users from the need to deal with much dreaded computer language code..."

PC Tech Journal: "Magic PC is probably the best integrated database application and screen generator that we have seen...very smooth system, and smoothness comes at a premium these days..."

PC World: "Relational data managers and application generators that offer power without programming are a bit like perpetual motion machines—very rare. Into that vacuum comes Magic PC, a data management tool without language that is ideal for turnkey applications..."

PC Week: "Rather than use a written programming language the user is given a great deal of freedom and power to create complex relational database applications...this package is a true time-saver..."

Get your Magic Tutorial for only $19.95

See for yourself how fast and powerful visual programming can be for you. Order your copy of **Magic PC Tutorial** including Tutorial disks, and a step-by-step tutorial to quickly set up an Order Entry application without coding. All for only $19.95 and We'll credit the Tutorial cost towards the $695 Magic PC price for up to 30 days.

Or save $500 at no risk!

Yes, for a short time only, qualified VAR's can save almost $500 off the list price and get the complete unprotected **Magic PC** software and documentation at a **special VAR price of $199*** with no risk. Keep it at this low price only if it's as good as we say, or return it within 30 days for a full refund if you're not completely satisfied. Act now and you'll also get 2 Magic Run (runtime modules) for only $95.

Order now for immediate delivery

Call this toll-free number now with your credit card or COD charge, or send the Order Coupon below today with your check to Aker.

1-800-345-MAGIC
in CA 714-250-1718

VISA

Yes, please rush me the following
Prices include 2nd day shipping

☐ Magic PC Tutorial	$ 25.95
☐ Magic PC (VAR's only)	$209.00
☐ Magic PC	$705.00
☐ Magic Run	$100.00
in CA add 6% tax	$____
Total	$____

Ship to: ________
Address: ________
City/ST/Zip: ________
Phone: ________

OEM inquiries are welcome. Prices valid in North America only.
*Less $19.95 restocking fee. Limit one per customer, subject to availability. Not for resale.

System requirements: IBM PC, XT, AT and 100% compatibles, PC-DOS 2.0 or later, 512K and hard disk.

AKER

Aker Corp. 18007 Skypark Cir. B2, Irvine, CA 92714
Tlx 4931184 AKER UI
Aker S.A. 11 Route de Florissant CH-1206, Geneve
Switzerland Tlx 421792 AKER CH

CIRCLE 369 ON READER SERVICE CARD

8088 Assembly-Language Programming Techniques

by Tom Disque

A collection of optimization techniques

Implementing a large software system on a personal computer requires that programmers recognize the bottlenecks of that system and reduce or eliminate them. It is not enough simply to recode a high-level language in assembly language; the code must take advantage of the special quirks of a host machine.

One of my responsibilities in the PC Host Group at SAS Institute is to ensure that PC host code is as small and as fast as possible. During this process, I have collected several optimization techniques, some of which are applicable to other architectures. These techniques are use of special instructions, unorthodox use of conventional instructions, and rearrangement of jump sequences. In the following discussion, please note that all timings are for the 8088 microprocessor and assume that the 4-byte 8088 prefetch queue is empty at the start of execution.

Repeat Instructions

One of the techniques involves using the repeat move and repeat store instructions. The idea is to move/store as much data per instruction as possible. Example 1, page 25, shows the code to move a number of bytes, where the source is in *ds:si*, the destination is in *es:di*, and the count is in *cx*. I have also applied this technique to the MC68000, although the even address requirements of that chip complicate the logic somewhat (see the box on page 26).

Another repeat instruction, the repeat compare, allows me to trim a few cycles. This instruction is used in a compare string routine that returns 0 if the strings are equal, 1 if string 1 < string 2, and -1 if string 1 > string 2. Originally, my code looked like that shown in Example 2, page 25, where string 1 is in *ds:si*, string 2 is in *es:di*, and the count is in *cx*. With this version, no matter which value is to be returned, a 16-cycle jump has to be executed. My final code is shown in Example 3, page 25. This code uses only nine cycles for above or below.

Probably the biggest bottleneck in any screen-intensive code for the IBM PC's color display is the wait for horizontal retrace—that period when the electron beam is turned off and moved to the far left of the screen to draw another scan line. This is the only time data can be moved to screen memory without causing flicker on the screen unless vertical retrace time is used (during vertical retrace, the electron beam is moved from the lower-right corner to the upper-left corner). Vertical retrace screen updates caused flickering during scrolls, so I used horizontal retrace screen updates. I discovered, however, that using a *movsw* for a character/attribute pair could cause flicker on some screens, so I changed the code in Example 4, page 25, to that in Example 5, page 25. The improved code enabled me to move one word per retrace on the IBM PC/XT and two words per retrace on the IBM PC/AT, with no flicker. I was even able to write three words per retrace with a small amount of flicker around the edges of the screen on the AT but decided to stick with two words.

Pointers

Much of our group's code involves pointer addition and subtraction, so we use sequences to add a constant to or subtract it from a normalized pointer and to produce a normalized pointer (normalized means the offset is always less than 16).

This code adds/subtracts the constant 1234H hex from the pointer in *ax:bx*:

Add

```
add   bx,0FFF4H
adc   ax,123H
and   bx,0FH
```

Subtract

```
sub   bx,0FFF4H
sbb   ax,123H
and   bx,0FH
```

We use the following sequences to add to or subtract from an unnormalized pointer, giving an unnormalized result with an offset less than 32,767:

Add

```
        add   bx,value
        jge   label1
        add   ax,800H
        and   bh,7FH
label1:
```

Subtract

```
        sub    bx,value
        jge    label2
        sub    ax,800H
        and    bh,7FH
label2:
```

We use the following sequence to normalize a pointer (again, in *ax:bx*):

```
mov    dx,bx
and    bx,0FH
shr    dx,1
shr    dx,1
shr    dx,1
shr    dx,1
add    ax,dx
```

Note that this sequence could be coded as:

```
mov    dx,bx
and    bx,0FH
mov    cl,4
shr    dx,cl
add    ax,dx
```

This version would be smaller, but the shifts would take longer on the 8086/80186, 80286, and so on. Note that because of the smaller size of the 8088/80188 microprocessors' prefetch queue, the time is approximately the same.

It is common in high-level languages such as C to assign a value to a variable based on a condition, as in:

```
i = 1;
if (k > 0) i = 2;
```

In this example, the idea is to assign the most likely value in the first statement and the least likely in the conditional. In some cases in assembly language, the reverse turns out to be most efficient, as follows:

```
        or     ah,80H     ; set blinking
        test   al,BLINK
        jnz    blinking
        and    ah,7FH     ; clear
                          blinking
blinking:
```

In this example, which sets or clears the blinking attribute for the screen display, the least likely alternative is for blinking to be set (not many people enjoy looking at a blinking screen eight hours a day).

If the jump is taken, it uses 16 cycles; if not, the jump uses 4 cycles and the instruction to clear blinking uses 4 cycles—half the amount of time of a jump taken. Note that the "taken jump/not taken jump" relative timings change with different chips in the 8080 family; the trend seems to be a reduction in the timing of a taken jump.

Miscellaneous Techniques

Finally, I have a few miscellaneous optimization techniques. When exchanging segment registers, a common technique is:

```
push   cs
pop    es
```

Much faster (4 cyles vs. 26 cycles) but bigger (4 bytes vs. 2 bytes) is:

```
mov    ax,cs
mov    es,ax
```

Another sequence to watch out for is:

```
        shr     cx,1            ;Convert byte count to word count
        jnc     word_move       ;If cx was odd, carry will be set
        movsb                   ;Move the odd byte
word_move:
        jcxz    exit            ;In case cx was equal to 1 originally
        rep     movsw           ;Move words
exit:
```

Example 1: *Using the repeat move and repeat store instructions to move a number of bytes*

```
        xor     ax,ax           ;Assume equal
        rep     compsb          ;Compare strings
        je      exit            ;If equal, we're finished
        ja      above           ;If above, set ax to 1
        dec     ax              ;Below.  Set flag to -1
        jmp     short exit
above:  inc     ax
exit:
```

Example 2: *Code for a compare string routine*

```
xor     ax,ax
rep     cmpsb
je      exit
sbb     ax,ax           ; If above, cf=0, if below cf=1
cmc                     ; If above, cf=1, if below cf=0
adc     ax,0            ; If above ax=1, if below ax=-1
```

Example 3: *Improved version of the compare string routine shown in Example 2*

```
lolab:  in      al,dx           ;Get status
        test    al,1            ;Is it low?
        jnz     lolab           ;Wait until it is
        cli                     ;No more interrupts
hilab:  in      al,dx           ;Get status
        test    al,1            ;Is it high?
        jz      hilab           ;Wait until it is
        movsw                   ;Write to the screen
        sti                     ;Reenable interrupts
```

Example 4: *Screen memory update routine synchronized to horizontal retrace*

```
        mov     bx,[si]         ;Load value "outside of" cli-sti
lolab:  in      al,dx           ;Get status
        test    al,1            ;Is it low?
        jnz     lolab           ;Wait until it is
        cli                     ;No more interrupts
hilab:  in      al,dx           ;Get status
        test    al,1            ;Is it high?
        jz      hilab           ;Wait until it is
        mov     es:[di],bx      ;Write to the screen
        sti                     ;Reenable interrupts
```

Example 5: *Improved version of code shown in Example 4*

```
mov     ax,word ptr [bp].value          ;8 bytes, 30 cycles
mov     bx,ax
mov     ax,word ptr [bp].value[2]

mov     ax,word ptr [bp].value[2]       ;6 bytes, 35 cycles
mov     bx,word ptr [bp].value

les     bx,[bp].value                   ;5 bytes, 35 cycles
mov     ax,es
```

Example 6: *Three ways to load a pointer into* ax:bx

```
          cmpi.l    #15,d0          If < 15, byte move is faster
          blt.s     bytemove
*                                   Special code to move words
*                                   Addresses must both be even or both be odd
          moveq.l   #0,d2           d2 is a flag for code below
          move.l    a0,d3           Copy the address register
          lsr.b     #1,d3           Is from an odd address?
          addx.l    d2,d2           If from is odd, increment flag

          move.l    a1,d3           Cannot 'btst' an address register
          btst      #0,d3
          beq.s     evenaddr
          addq.l    #1,d2           To is odd. increment flag

evenaddr  btst      #0,d2           If one addr is odd and the other even,
          bne.s     bytemove        we cannot do it

          lsr.b     #1,d2           If both were odd, d2 is 2; else it is 0
*                                   Now d2 = 1 indicates odd, d2 = 0 says even
          btst      #0,d0           Is n an odd number?
          beq.s     evenlen
          addq.l    #1,d2           N is odd. Set flag

evenlen   cmpi.l    #1,d2           Find out which even/odd
*                                    combination we have here
          beq.s     oddeven         One is odd, one is even
          blt.s     wordmove        Both are even. Take off!
          move.b    (a0)+,(a1)+     Both are odd. Fix the odd byte

wordmove  asr.l     #2,d0           Convert byte count to word count
          bcc.s     longmove        Any odd byte has been moved
          move.w    (a0)+,(a1)+     Now move extra half-word

longmove  subq.l    #1,d0           Decrement for dbf
longloop  move.l    (a0)+,(a1)+
          dbf       d0,longloop
          bra.s     exit

oddeven   btst      #0,d0           Was the count the odd one?
          bne.s      oddcnt

evencnt   subq.l    #1,d0           The address was odd, count even;
          move.b    (a0)+,(a1)+     Now the addr is even, count odd!

oddcnt    asr.l     #2,d0           Convert byte count to word count
          bcc.s     lngmove2        The odd byte will been moved later
          move.w    (a0)+,(a1)+     Now move extra half-word


lngmove2  subq.l    #1,d0           Decrement for dbf
lngloop2  move.l    (a0)+,(a1)+
          dbf       d0,lngloop2
          move.b    (a0)+,(a1)+     Move the odd byte
          bra.s     exit

bytemove  subq.l    #1,d0           Decrement for dbf
loop      move.b    (a0)+,(a1)+     *to++ = *from++
          dbf       d0,loop         while --n > 0

exit      rts                       Return
```

Example 7: *Performing 32-bit moves on the 68000*

8088 ASSEMBLY LANGUAGE
(continued from page 25)

```
mov   cl,4
shl   ax,cl
```

Faster (8 cycles [plus 8 more to fetch the two extra instructions] vs. 28 cycles) but bigger (8 bytes vs. 4 bytes) is:

```
shl   ax,1
shl   ax,1
shl   ax,1
shl   ax,1
```

If the count had been 8:

32-Bit Moves on the Motorola 68000 Microprocessor

The Motorola 68000 microprocessor poses a somewhat more difficult problem than the Intel chips when you want to move more than 8 bits of data at a time. The 68000 can move 32 bits in a single instruction, but it must move from an even address when doing so; only 8-bit moves can have an odd address.

Example 7, left, shows code for a technique that allows 32-bit moves most of the time. Here, register *A0* contains the pointer from which the data is to be moved, register *A1* contains the register to which the data is to be moved, and register *D0* contains the number of bytes to move. The overhead because of the length check over a straight byte move (at 8 bytes) is 8.6 percent; the overhead if one address is odd and the other is even (at 16 bytes) is 4.9 percent. In order to have the best of both worlds, I wrote separate 8-byte and 16-byte move routines that only move bytes at a time. Because these are the only common sizes below or close to the threshold that I move, I have gained an increase in speed at every level. In fact, moves of as little as 100 bytes produce a threefold increase in speed over the simple byte move routine.

WINDOWS FOR DATA™

The first choice of professional C programmers

"Windows for Data is the best programming tool I've ever used.

It's the most flexible I've seen.
Whenever I've wanted to do something,
I've been able to find a way."

Steven Weiss,
Stratford Systems

Professionals choose our tools because they are designed, crafted, and supported for professionals. Here at Vermont Creative Software, we understand that performance and pleasure in programming derive from more than a long list of functions. **Windows for Data** provides:

PROFESSIONAL FLEXIBILITY: Our customers repeatedly tell us how they've used WFD in ways we never imagined - but which we anticipated by designing WFD for unprecedented adaptability. Virtually every capability and feature can be modified to meet special needs. You will be amazed at what you can do with WFD.

PROFESSIONAL PERFORMANCE: Screen output is crisp and fast. Windows, menus, and data-entry forms snap up and down from the screen. WFD is built upon and includes **Windows for C, the windowing system rated #1 in speed and overall quality** in PC Tech Journal (William Hunt, July 1985).

PROFESSIONAL RELIABILITY: An unreliable tool is worse than no tool at all. VCS products are known in the industry for their exceptional reliability. Ask anyone who owns one.

PROFESSIONAL DOCUMENTATION: Over 600 pages of documentation provide step-by-step explanations for each major application, a reference page for each function, listings of functions alphabetically and by usage, and a fully cross-referenced index. Extensive tutorials and demonstration programs assist learning.

PROFESSIONAL TECHNICAL SUPPORT: The same expert programmers that develop our products provide prompt, knowledgeable technical support.

PROFESSIONAL PORTABILITY: High-performance versions of VCS products are available for XENIX, UNIX, and VMS, as well as DOS. No royalties on end-user applications.

OUR CHALLENGE AND GUARANTEE

If you have an application where no other tool can do the job, try **Windows for Data.** If it doesn't help you solve your problem, RETURN FOR A FULL REFUND. YOU MUST BE SATISFIED.

Ask for **FREE DEMO DISKETTE**

Vermont Creative Software

21 Elm Ave.
Richford, VT 05476
Telex: 510-601-4160 VCSOFT
Tel.: 802-848-7731

Prices: PCDOS* $395: XENIX, VMS, UNI
*PCDOS specify C compiler.

WINDOWS FOR DATA

for DOS, UNIX, VMS ...

The complete windowing data entry, menu, and help system that does the hard job others can't — we **guarantee** it!

Pop-up data entry windows; field types for all C data types, plus decimals, dates, and times; auto conversion to and from strings for all field types; system and user supplied validation functions; range checking; required, must-fill, and protected fields; free-form movement; multiple-choice field entry; scrollable sub-forms. Branch and nest windows, forms, and menus.

Complete context-sentitive help system with pop-up windows and scrollable text.

Pop-up, pull-down, scrollable, and Lotus-style menus.

NEW FOR DEBUGGING: Exclusive **VCS Error Traceback System** automatically identifies the location and cause of program errors. Eliminates the need to code error checks on all function calls! **VCS Memory Integrity Checking** helps catch those hard-to-detect, memory-corruption errors.

NEW FOR ERROR HANDLING: Install your own error handler to be called whenever a function detects an error.

NEW FORM LAYOUT UTILITY simplifies form design.

CIRCLE 157 ON READER SERVICE CARD

(continued from page 26)

```
xor     al,al
```

would be even faster and the same size. An optimization of structure references replaces:

```
xor     si,si
             ;Reference the 0th structure
mov     ax,[si].value
```

with:

```
mov     ax,[0].value
```

A pointer can be loaded into *ax:bx* in three ways, as shown in Example 6, page 26. Note that the first method is faster because loads and stores using the accumulator don't use any cycles for effective address calculation. On the 80188 microprocessor and later chips, the effective address calculation is taken care of by the hardware, which means that the third method will be fastest on those chips.

One quirk of the 8088 microprocessor instruction set appears in the timing of the *movsw* instruction as opposed to the same instruction prefaced by a repeat byte. The *movsw* instruction is 26 cycles alone and 9 + 25 cycles per repeat when prefaced by a repeat byte. This means that when less than nine repetitions are to be done, the faster alternative is to code the *movsw* instructions one at a time. The same is true of the *movsb*, *stosw*, and *stosb* instructions. This seems odd because the repeat instructions always have the overhead of decrementing the *cx* register; thus, you'd expect it to be slower.

It is always important to check the timings of instruction sequences on the target machine; the relative magnitudes of the timings given here, for instance, are different from those for the 80188 microprocessor. I have used the 8088 microprocessor's timings because the 8088 shows improvements most dramatically. Common assumptions about floating-point arithmetic don't always apply on the 8087 coprocessor, either. For example, one common assumption is that, if three adds could replace two multiplies, the resultant code would be faster—not so with the 8087 coprocessor!

I have shown that code can be optimized by use of special instructions, unusual usage of conventional instructions, and rearrangement of code. The techniques outlined here, if used alone and in infrequently called code, do not necessarily produce noticeable results, however. First, you must identify bottlenecks—even a small reduction in time is noticeable in a true bottleneck. If you don't notice any speed improvements, it's not worth optimizing your code.

Acknowledgments

I would like to thank John Toebes and Mike Jones for the use of their pointer techniques in this article.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 2**.

```
        shr     cx,1            ;As in the text
        jnc     word_move
        movsb
word_move:
        jcxz    exit
        mov     ax,si           ;Use ax to test addresses
        and     ax,di           ;Put the addresses together
        shr     ax,1            ;If both addresses were odd,
        jnc     not_odd         ;the carry flag will be set
        movsb                   ;Move to an even address
        dec     cx              ;One less word for the repeat
        jcxz    finish          ;If only one word
        rep     movsw
finish: movsb                   ;Move the last byte
        jmp     short exit
not_odd:
        rep     movsw
exit:
```

Example 8: *Correction to increase speed of* movsw *code for odd addresses on an 8086*

An Improved Move for Wider Buses

The repeat move given in the text is optimal for machines using an 8-bit data path, but the code can be improved in some cases for machines using a 16-bit path (that is, using CPUs such as the 8086, 80186, and 80286). When a *movsw* instruction executes on the 8088, it generates two external bus cycles regardless of whether the addresses are odd or even. When a *movsw* executes on the 8086, however, only one bus cycle is generated for even addresses. (Two bus cycles are still generated for odd addresses.)

At first glance, the code to convert from byte count to word count in the text might seem useless. But bus cycles alone do not determine instruction speed. Empirical results on a Leading Edge Model D (with an 8088 CPU) show roughly a 20 percent increase in execution speed. Ideally, you would expect the speed to be almost double, so you can see what a large effect the bus size has on execution speed when accessing memory.

In order to correct for the odd address problem, the code sequence in Example 8, above, can be used.

Without this correction, the *movsw* code for even addresses is twice as fast as for odd addresses; in fact, the odd address word move runs as slow as a byte move! With the correction, the odd address' move is almost the same speed. Please note that this code is best used for moving large blocks of data; for small blocks, the overhead is a significant factor. Also keep in mind that both pointers will usually be even because most compilers try to align objects on the most efficient boundaries.

Logic and Knowledge Representation in PROLOG

by Richard Butrick

The syntax of PROLOG is essentially limited to fact statements, which are simple noncompound sentences, and rule statements, which are *if* statements with the consequent, which must be a fact statement, placed on the left. The form of the rule statement is *head if body*. The *head* must be a simple noncompound statement, and the *body* can be either simple or a compound built up of conjunctions, disjunctions, and negations. The body cannot itself be a conditional containing *if*.

Even Procrustes would find this a severe limitation on human expression. Once you get past the cozy genealogical examples on which PROLOG texts rely to introduce logic programming, reformulation of human thinking into forms pallatable to PROLOG can get pretty rough. Moreover, weighty considerations concerning SLD-resolution (which stands for Selecting a literal, using a Linear strategy, and searching the space of possible deductions Depth-first) and unsatisfiable sets of Horn clauses aside, PROLOG's arsenal of deductive weapons consists of a rapid fire peashooter known to the scholastics as *modus ponens*. Thus casting your pearls of wisdom into the jaws of PROLOG syntax does not guarantee even the most obvious deductions being drawn by PROLOG's single-cylinder inference engine.

Consider Lao Tsu's chestnut: "When opposites supplement each other, everything is harmonious."

Richard Butrick, Ohio University, Athens, OH 45701. Richard is a professor in the Computer Science Department.

Reformulation of human thinking into PROLOG can be rough.

This can be cast into PROLOG syntax rather handily as *Everything_is_harmonious if all_opposites_supplement _each_other*. PROLOG, however, cannot even define from this that *not all_opposites_supplement_each_other*, given the fact that *not everything_is_harmonious*. Basically, the only inference that PROLOG can perform (extended by unification) is to infer *R* from:

R if S1 and S2 and S3 . . . and Sn

and:

S1 and S2 and S3 . . . and Sn

The *R* on the left cannot even be compound! This means that the *R* on the left cannot be of any of the following forms: *not L*, *L or M*, *L and M*. What then do you do with "If there is a sharp move in the market, then it is either short covering on the up side or a purely technical reaction on the down side" (that is, *C or R if M*)? Or even, more simply, "If it is an ordinary market, then it is not wise to buy in the first hour" (that is, *not W if O*)?

Knowledge representation in PROLOG comes down to formulating sentences in a syntax acceptable to PROLOG and in formulating the sentences in such a way that PROLOG will make the appropriate deductions. A lot of creative thinking is involved in coming up with the sentences to be represented in PROLOG (knowledge codification) and in organizing blocks of knowledge (modularization), but it still comes down to knowing how to enter such sentences to assure the level of deductive completeness desired. Two categories of problems have to be dealt with: compound fact/consequent representation problems and existential quantification representation problems.

Compound Fact/Consequent Representation

As indicated, the official syntax of PROLOG only allows the introduction of two types of sentences into a program. It accepts simple or noncompound statements (statements that do not contain any *if*s, *and*s, *or*s, or *not*s), and it accepts conditionals of the form *noncompound if simple-or-compound*. The compound side of the conditional cannot itself be a conditional (it can contain only *and*s, *or*s, and *not*s). In practice, there are several ways around this restriction, and in fact it is crucial to get around it for full knowledge representation.

The points made here are with specific reference to the Simple syntax of micro-PROLOG, which is the syntax that is closest to classical logic and to English syntax. Considerations brought forward apply equally to C&M syntax, however, because they refer to the underlying logic of PROLOG based on SLD-resolution. I might also mention in passing that, among the better-known, interpreted PROLOGs, only micro-PROLOG imple-

ments tail recursion correctly with no limit on the recursion. Moreover, metalevel programming involving standard syntax can be incorporated into Simple programs, and in that sense Simple syntax is a full-fledged PROLOG system and not a Simple Simon syntax.

Negative Facts

There are three ways around the restriction on negative literals (negations of simple sentences):

fact if FAIL (1)
not_fact (2)
fact(not) (3)

For example, the following PROLOG program snippet behaves just as if it contained negative facts in its fact-rule base:

might_be_bear_market if not
bull_market (1a)
bull_market if FAIL (1b)

PROLOG responds to the query *is(bull_market)* with *no* and to the query *is(might_be_bear_market)* with *yes*.

The corresponding program for the *not_fact* solution is:

might_be_bear_market if
not_bull_market (2a)
not_bull_market (2b)

This version can make the deduction *might_be_bear_market* and, obviously, *not_bull_market*, but it cannot make the deduction *not bull_market*. In this regard, solution (2) is weaker than solution (1).

The third solution is really a sleazy solution that only works because the dictionary maintained by the syntax analyzer does not keep the degree of the predicate:

might_be_bear_market if not
bull_market (3a)
bull_market(not) (3b)

Thus PROLOG responds to the query *is(bull_market)* with *no* even though *bull_market* was used as a one-place predicate and not as a fact (zero-placed predicate). To make matters worse, Simple syntax accepts the postfix notation and so (3b) could have been *not bull_market*. Rather than *not* being treated as negation, however, it is treated as a thing that has the property *bull_market*. These conceptual confusions make little difference, however, as the correct responses and deductions will be made. The net effect is that of declaring *bull_market* to be a *data-rel*, which enters it in the dictionary but not the database.

It might seem at this point that solution (1) works fine and that negative facts are neatly handled by the built-in primitive *FAIL*. Unfortunately, there is a serious problem lying behind the negative-fact problem and that is that PROLOG will make fallacious inferences with the introduction of the built-in *not*. Consider the following example:

The incorporation of recursion search methods within a deductive framework makes PROLOG valuable and powerful.

sentence_1 if not sentence_2
sentence_2
sentence_3 if not sentence_1

PROLOG not only answers *no* to *is(sentence_1)* but it also deduces *sentence_3*. Let us hope the following sentences do not find their way into a Pentagon expert system:

a_pre-emptive_strike-will_occur if
not supreme_caution_exercised
supreme_caution_exercised
pre-emptive_strike_defence_unnecessary if not pre-emptive_strike _will_occur

Yes will be the response to the query *is(pre-emptive_strike_defence_unnecessary)*. Now there is a piece of artificial intelligence. The negation logic for PROLOG operates under what is called negation-by-failure, or the closed-world principle. This means that if a sentence S is not deducible, then not S is deducible. This type of argument is known in logic texts as the argument from ignorance, and it commits the fallacy of failing to distinguish between known truth and truth. In a complete system everything true is asserted to be true either directly or by implication. Hence that which is not asserted is false. Because complete systems are few and far between, and no system using elementary arithmetic is complete (this is known as Gödel's Incompleteness Result), what is needed is a form of negation for incomplete systems. For such systems, the built-in *not* should be used only in those cases in which failure to succeed implies falsity, as for example in *not x ON (a b c d)*. Otherwise, the negation logic needed must be provided by the program. This is solution (2).

Compound Facts

Consider the following compound:

if sentence_1 then sentence_2 or sentence_3

The attempted representation in PROLOG as:

sentence_2 or sentence_3 if sentence_1

does not work because a compound cannot occur to the left of an *if*. The question of representation turns on the sorts of deductions that could be made from the sentence. The following are possible deductions: *not_sentence_1, sentence_2 or sentence_3, sentence_2, sentence_3*. The deduction of *sentence_3*, for example, would require the following entry:

sentence_3 if sentence_1 and not_sentence_2

Table 1, page 32, gives some useful conversions. Which conversion or conversions you use depends on which sentences you target as possible goals within the knowledge base. You must then ensure that you use a consistent representation within the entire knowledge base. Needless to say, this is no mean task, and generally speaking, unless all the sentences to be represented fit within the official syntax, attempting to ensure that all legitimate deductions can be drawn is an impossible task.

(continued from page 31)

The following is an example of a knowledge representation problem at the level of sentence logic (as opposed to quantificational- or predicate-level logic):

1. If aggregate expenditure is unresponsive to changes in the money supply and unresponsive to changes in the interest rate, then monetarism offers little hope for controlling the economy. (if AUM and AUI then MLH)
2. If there is a large infusion of money, then interest rates will fall and people will be induced to hold money upon a large infusion of money rather than invest or spend. (if LIM then IRF and HLM)
3. If people are induced to hold money upon a large infusion of money, then aggregate expenditure is unresponsive to changes in the money supply. (if HLM then AUM)
4. Because it cannot be the case that both interest rates fall and people are not induced to hold new money, then aggregate expenditure is unresponsive to changes in the interest rate. (if either not IF or HNM then AUI)

Representation of these "rules" in PROLOG depends on the sorts of deductions that are of potential interest. Full representation is prohibitively large. The following representation allows the principle deductions of *MLH* or of *not_AUM*, depending on the facts added to the rule base:

MLH if AUM and AUI
not_AUM if not_MLH and AUI
IRF if LIM
not_LIM if not_IRF
HLM if LIM
not_LIM if not_HLM
AUM if HLM
AUI if not_IF
AUI if HNM

Given the facts *LIM* and *not_IF*, PROLOG can deduce *MLH*. Given the facts *not_MLH* and *HNM*, PROLOG can deduce *not_AUM*, and so forth.

Logic	PROLOG
not L	not_L
L and M	L M
if L then M	M if L not_L if not_M
L or M	L if not_M M if not_L
not(L and M)	not_M if L not_L if M
if L then M and K	M if L not_L if not_M K if L not_L if not_K
if L then M or K	M if L and not_K K if L and not_M not_L if not_M and not_K
if L or M then K	K if L K if M not_L if not_K not_M if not_K
if L and M then K	K if L and M not_M if not_K and L not_L if not_K and M
if L or M then K or H	K if L and not_H K if M and not_H H if L and not_K H if M and not_K not_L if not_K and not_H not_M if not_K and not_H
if L and M then K or H	K if L and M and not_H H if L and M and not_K not_L if not_K and not_H and M not_M if not_K and not_H and L

Table 1: *Some useful conversions between logic and PROLOG*

Representation at the Quantificational Level

Variables in simple sentences and variables that occur on both the left and right of a conditional are treated as universally quantified. Variables that occur only on the right side of the *if* are treated as existentially quantified. Although this sums up in a nutshell PROLOG's treatment of quantification, a great deal needs to be done to explain this policy and how to work within the constraints of the policy.

In PROLOG the sentences:

x is_pernicious
x is_pernicious if x is_devious
x is_pernicious if y is_devious and x admires y

are treated as if they were the following:

Every x is pernicious
(Everyone [at Harry's Place] is pernicious)
Every x is pernicious if that x is devious
(Everyone who is devious is pernicious)
Every x is pernicious if some y is devious and x admires that y
(Anyone who admires a devious person is pernicious)

PROLOG provides no explicit notation to indicate whether an *x* in a sentence is supposed to mean "some x,"

EVEN MORE POWER AND FLEXIBILITY

BRIEF 2.0

Users and industry press alike have unanimously proclaimed BRIEF as the best program editor available today. Now, the best gets better, with the release of BRIEF 2.0.

Straight from the box, BRIEF offers an exceptional range of features. Many users find that BRIEF is the only editor they'll ever need, with features like real, multi-level Undo, flexible windowing and unlimited file size. But BRIEF has tremendous hidden power in its exclusive macro language. With it, you can turn BRIEF into your own custom editor containing the commands and features you desire. It's fast and easy.

Jerry Pournelle, columnist for BYTE magazine summed it all up by saying BRIEF is, **"Recommended. If you need a general purpose PC programming editor, look no further."** His point of view has been affirmed by rave reviews in C JOURNAL, COMPUTER LANGUAGE, DR. DOBB'S JOURNAL, DATA BASED ADVISOR, INFOWORLD AND PC MAGAZINE.

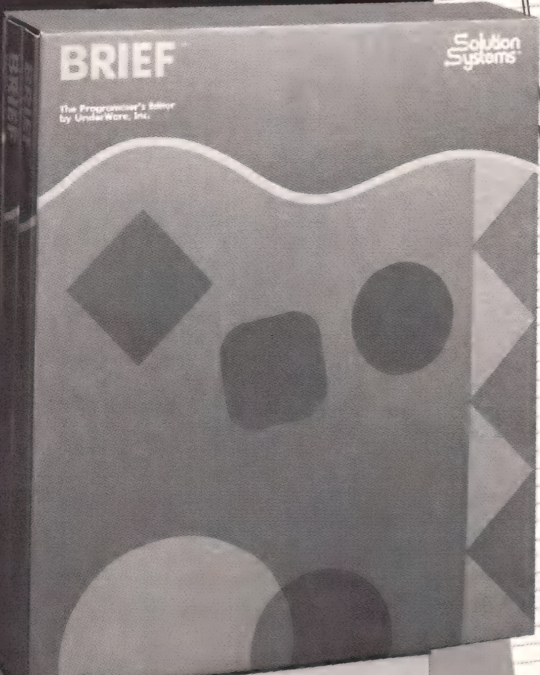
One user stated "BRIEF is one of the few pieces of software that I would dare call a masterpiece." Order BRIEF now and find out why. **BRIEF 2.0 is just $195.** If you already own BRIEF, call for upgrade information.

TO ORDER CALL: 1-800-821-2492
(in MA call 617-659-1571)

As always, BRIEF comes with a 30 day money-back satisfaction guarantee.

Solution Systems™

335-D Washington St.
Norwell, MA 02061
(617) 659-1571



Look at these BRIEF 2.0 enhancements!

Main Features:

- All new documentation with tutorials on basic editing, regular expressions **and** the BRIEF Macro Language.
- Setup program for easy installation and configuration. (Requires no knowledge of the macro language)
- Increased speed for sophisticated operations like Undo and Regular Expression Search.
- Expanded regular expressions, with matching over line boundaries.
- More block types, with marking by character, line or column.
- Command line editing (move cursor, add and delete characters, specify command parameters).
- Support for more programming languages.
- Optional borderless windows.
- Enhanced large display support, including wider displays.
- Reconfigurable indenting for C files (supports most indenting styles).

Plus the basic features that made BRIEF SO popular!

Basic Features:

- Full multi-level Undo
- Windows
- Edit many files at once
- File size limited only by disk space
- Automatic language sensitive indentation

Requires an IBM PC or compatible with at least 192K RAM.
BRIEF is a trademark of UnderWare, Inc.
Solution Systems is a trademark of Solution Systems.

CIRCLE 142 ON READER SERVICE CARD

Logic	PROLOG
(Ex)Fx (There is an x, Fx) (Something is an F)	`F(alpha)`
(Ex)Gx	`G(beta)`

Table 2: *Representing existing but unknown objects in* PROLOG

Logic	PROLOG
(x)(Ey)x < y (Every no. is less than some no.)	`Less(x (alpha x))`

Table 3: *Representing dependencies among unknown objects in* PROLOG

PROLOG SEMANTICS
(continued from page 32)

or "an x," or "all x." Expressions such as "all," "some," and "any," which are used to indicate quantification, are simply not part of PROLOG's vocabulary. How then do you say that there are some pies in the sky as opposed to saying that x is a pie in the sky and have PROLOG treat this statement as if it meant that every x is a pie in the sky? To a large extent the logic for indicating "some" (existentially quantified variables) must be provided by the programmer.

Thoralf Skolem, the great Norwegian logician, is credited with providing the first systematic method for representing quantifiers without using quantifiers. His method utilizes what are aptly called Skolem functions. Fortunately, his ideas are intuitive enough to avoid having to refer directly to his formal presentation. You might feel inclined at this point to inquire politely, without denigrating the wonders of Skolemizing quantifiers, why you would wish to represent quantificational relationships without using quantifiers. The answer is that *modus ponens*, the heart of the PROLOG inference engine, can't work with quantifiers. Besides, logicians love to frighten their hapless dinner companions by telling them, as if indifferent to moral outrage, that they have spent the day in their office Skolemizing quantifiers. But then, quantifiers will do anything for a *modus ponens*.

To say that something is an F is not to say that any specific thing is an F. Thus in a domain of three things— {a b c}— you cannot infer, say, that c is an F because something is an F. One of them is an F, but which one is, is not known. The idea here is to invent a fictitious name that by convention is not a name of anything in the domain of discourse. This corresponds to the standard mathematical practice that runs as follows: "Something is an F. Call it alpha." Giving a fictitious name or moniker to unknown persons is also common in ordinary language—for example, "Kilroy" or "Jack the Ripper." From a logical point of view, the important thing is, first of all, to distinguish between the name of an unknown object and the name of a known object. Using Greek

letters is a convenient way of doing this. The second thing is to make sure that different Greek letters are assigned to objects whose existence is asserted by different "somes" (existential quantifiers), as shown in Table 2, page 34.

It is important to block the inference that something is both an F and a G from something is an F and something is G. You do this by using different Greek letters that may or may not refer to the same object. The following PROLOG statements provide the information necessary for PROLOG to make the inference that *alpha* and *beta* have at least one property in common:

```
F(alpha)   (* There are Fs *)
G(beta)    (* There are Gs *)
G(x) if F(x) (* All Fs are Gs *)
```

Given the query *is(G(alpha))*, PROLOG answers *yes*.

When existential quantifiers are used with universal quantifiers, the logic for eliminating quantifiers becomes more complicated. Thus "every number is less than some number" cannot be rendered as "every number is less than alpha." Clearly the intention of the first statement is not that every number is less than one and the same number, alpha. This would make alpha less than itself! The intention is that the alphas are different for different numbers. This is done by making alpha x-dependent, as shown in Table 3, page 34. Instead of using alpha, the term *(alpha x)* is used. The variable dependencies for alpha are only those universally quantified variables that precede it when written in quantified form (see Table 4, above).

To get an idea of the inferences PROLOG can make with these sorts of sentences, consider the following:

```
Loves(alpha x)    (* Someone loves everyone *)
Loves(x (beta x))   (* Everyone loves someone *)
```

To each of the queries *is(Loves(alpha alpha))*, *is(Loves(alpha (beta x)))*, and *is(Loves((beta x) (beta (beta x))))*, PROLOG responds *yes*.

Unfortunately, PROLOG doesn't directly make the legitimate inference "everyone is loved by someone"

Logic	PROLOG
(x)(Ey)Lxy	`Loves(x (alpha x))`
(x)(Ey)Lyx (Everyone is loved by someone)	`Loves((alpha x) x)`
(Ex)(y)Lxy (Someone loves everyone)	`Loves(alpha y)`
(Ex)(y)Lyx (Someone is loved by everyone)	`Loves(y alpha)`

These sentences are taken as isolated examples. Hence the use of the same Greek letter in all the examples rather than using new letters for each existential quantifier.

Table 4: *Resolving ambiguities of affection in* PROLOG

DR. DOBB'S JOURNAL

OF SOFTWARE TOOLS

THE R_x FOR PROGRAMMERS

SUBSCRIBE NOW AND SAVE OVER 15% OFF THE NEWSSTAND PRICE!

SUBSCRIBE & SAVE

$5 SAVINGS

Subscribe to DR. DOBB'S JOURNAL and save over $5—a 15% savings off the cover price!

☐ Please charge my: ☐ Visa ☐ MasterCard ☐ American Express

☐ Payment enclosed ☐ Bill me later

Card # ____________ Exp. date ____________

Signature ____________

Name ____________

Address ____________

City ____________ State ______ Zip ______

ONLY $29.97! YOU SAVE OVER $5.00

Savings based on a full one-year cover price of $35.40. Canada and Mexico add $10 for surface mail per year. All countries add $27 for airmail per year. All foreign subscriptions must be prepaid in U.S. dollars drawn on a U.S. bank. Please allow 6-8 weeks for delivery of first issue.

A Publication of M & T Publishing, Inc. 3450

SUBSCRIBE & SAVE

$5 SAVINGS

Subscribe to DR. DOBB'S JOURNAL and save over $5—a 15% savings off the cover price!

☐ Please charge my: ☐ Visa ☐ MasterCard ☐ American Express

☐ Payment enclosed ☐ Bill me later

Card # ____________ Exp. date ____________

Signature ____________

Name ____________

Address ____________

City ____________ State ______ Zip ______

ONLY $29.97! YOU SAVE OVER $5.00

Savings based on a full one-year cover price of $35.40. Canada and Mexico add $10 for surface mail per year. All countries add $27 for airmail per year. All foreign subscriptions must be prepaid in U.S. dollars drawn on a U.S. bank. Please allow 6-8 weeks for delivery of first issue.

A Publication of M & T Publishing, Inc. 3450

COMMENTS & SUGGESTIONS

Dear Reader, **July 1987, #129**

Dr. Dobb's has a long tradition of listening to its readers. We like to hear when something really helps or, for that matter, bothers you. In this hectic world of ours, however, it is often difficult to take the time to write a letter. This card provides you with a quick and easy way to correspond and, if you include your name and address, we may use appropriate comments in The Letters column. Simply fill it out and drop it in the mail. —Ed

Which articles or departments did you enjoy the most this month? Why?

____

Comments or suggestions: ____________

Name: ____________

Address: ____________

No Postage
Necessary
If Mailed
In The
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT 790, REDWOOD CITY, CA

Postage Will Be Paid By Addressee

Dr. Dobb's Journal of
Software Tools
Box 3713
Escondidio, CA 92025-9843

(continued from page 35)

(*Loves((delta x) x)*) from "someone loves everyone" (*Loves(alpha x)*). It might be argued that in effect it makes the inference because the query "is everyone loved by someone?" could be said to take the form *which(x: Loves(y x))*. The answer will be *x* , and this could be taken as meaning "everyone is loved by someone." The *y* in the query is understood by PROLOG as being existentially quantified.

Some Standard Quantificational Statements

It is typical of human discourse to identify two classes of objects and claim some sort of relationship between the the two classes. Those sentences that advance all-some or some-all relationships have particularly interesting logical properties and illustrate the use of Skolem functions. Consider the sentence "All passengers were questioned by customs officers." This statement identifies two classes of objects—passengers and customs officers—and asserts that each of the former was searched by at least one (not necessarily one and the same) of the latter. This differs from the statement "Some customs officers searched all of the passengers," but a relationship is still being asserted between two classes of objects. Table 5, above, considers variations of these "dual-class" sentences.

The following is a well-known test problem for formulation into an inference system based on *modus ponens* and unification:

The customs officials searched everyone who entered the country who was not a VIP.
Some of the drug pushers entered this country and they were only searched by drug pushers.
No drug pusher was a VIP.

This problem is to be represented in such a way as to support the inference that some of the officials were drug pushers. The representation in PROLOG is as follows:

C((alpha x)) if E(x) and not_V(x)
S((alpha x) x) if E(x) and not_V(x)
D(beta)

Logic	PROLOG
(x)(Px --> (Ey)(Cy & Syx)) (Every passenger was searched by some customs officer)	`S((alpha x) x)) if P(x)` `C((alpha x)) if P(x)`
(Ex)(Cx & (Y)(Py --> Sxy)) (Some customs officer searched every passenger)	`C(alpha)` `S(alpha y) if P(y)`
(Ex)(Px & (y)(Cy --> Syx)) (Some passenger was searched by every customs officer)	`P(alpha)` `S(y alpha) if C(y)`
(x)(Cx --> (Ey)(Py & Sxy)) (Every customs official searched a passenger)	`S(x (alpha x)) if C(x)` `P((alpha x)) if C(x)`

These are treated as isolated sentences, so alpha is used in all cases instead of using new Greek letters for each new existential quantifier. The PROLOG sentences are subject to the truth functional conversions indicated in the first section of this article (compound fact/consequent representation problems).

Table 5: *Representing relationships among classes of objects in PROLOG*

PROLOG SEMANTICS
(continued from page 37)

E(beta)
D(y) if S(y beta)
not_V(x) if D(x)

To the query *is(S((alpha beta) beta))*, PROLOG answers *yes*, and to *which(x: C(x) and D(x))*, PROLOG responds *(alpha beta)*.

With the use of Skolem functions, quantificational-level logic can be worked into the patois of PROLOG. But this is only first-order quantificational logic without identity. Introducing identity, whereby S(y) is inferred from S(x) and x = y, is another story in itself and requires second-order programming.

Summary Remarks

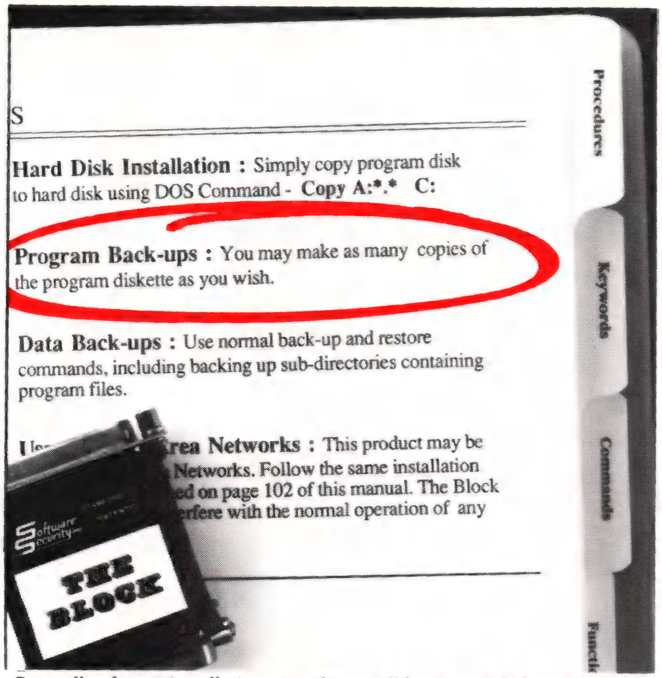
From a logical point of view, attempting to make all deductions on the basis of *modus ponens* and unification seems a needless and stultifying limitation on human reasoning capacity. Indeed, deduction engines that employ a full complement of inference rules are legion in academe. But they suffer from several problems, not the least of which is speed or lack thereof. The killer problem is that of incorporating recursion with deduction. PROLOG accomplishes this, and the incorporation of recursion search methods within a deductive framework is really what makes PROLOG valuable and powerful. Its logical limitations are the price paid for this incorporation, but the trade-off is likely to seem a worthwhile one for some time.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

"How to protect your software by letting people copy it."

By Dick Erett, President of Software Security



Soon all software installation procedures will be as straightforward as this. The only difference will be whether you include the option to steal your product or not.

Inventor and entrepreneur, Dick Erett, explains his company's view on the protection of intellectual property.

"A crucial point that even sophisticated software development companies and the trade press seem to be missing or ignoring is this:

Software protection must be understood to be a distinctively different concept from that commonly referred to as copy protection.

Fundamentally, software protection involves devising a method that prevents unauthorized use of a program, without restricting a legitimate user from making any number of additional copies or preventing program operation via hard disk or LANs.

Logic dictates that magnetic media can no more protect itself from misuse than a padlock can lock itself.

Software protection must reside outside the actual storage media. The technique can then be made as tamper proof as deemed necessary. If one is clever enough, patent law can be brought to bear on the method.

Software protection is at a crossroads and the choices are clear. You can give product away to a segment of the market, or take a stand against the theft of your intellectual property.

"... giving your software away is fine..."

We strongly believe that giving your software away is fine, if you make the decision to do so. However, if the public's sense of ethics is determining company policy, then you are no longer in control.

We have patented a device that protects your software while allowing unlimited archival copies and uninhibited use of hard disks and LANs. The name of this product is The BLOCK™.

The BLOCK is the only patented method we know of to protect your investment. It answers all the complaints of reasonable people concerning software protection.

In reality, the only people who could object are those who would like the option of stealing your company's product.

"... eliminating the rationale for copy-busting..."

Since The BLOCK allows a user to make unlimited archival copies the rationale for copy-busting programs is eliminated.

The BLOCK is fully protected by federal patent law rather than the less effective copyright statutes. The law clearly prohibits the production of work-alike devices to replace The BLOCK.

The BLOCK attaches to any communications port of virtually any microcomputer. It comes with a unique customer product number programmed into the circuit.

The BLOCK is transparent to any device attached to the port. Once it is in place users are essentially unaware of its presence. The BLOCK may be daisy-chained to provide security for more than one software package.

Each software developer devises their own procedure for accessing The BLOCK to confirm a legitimate user. If it is not present, then the program can take appropriate action.

"... possibilities... limited only by your imagination..."

The elegance of The BLOCK lies in its simplicity. Once you understand the principle of The BLOCK, hundreds of possibilities will manifest themselves, limited only by your imagination.

Your efforts, investments and intellectual property belong to you, and you have an obligation to protect them. Let us help you safeguard what's rightfully yours. Call today for our brochure, or a demo unit."

Software Security inc.

870 High Ridge Road Stamford, Connecticut 06905
203 329 8870

CIRCLE 170 ON READER SERVICE CARD

Multitasking with Turbo Pascal

by Craig A. Lindley

How to add a round-robin task scheduler with a few short procedures

Multitasking is the ability to distribute the resources of a single CPU to several processes or tasks. This leads to better utilization of a CPU: while one process is stuck waiting for an event to occur before it can continue (a response from the user, for example), another process can continue running. The CPU can thus perform more useful work with greater efficiency. To implement multitasking requires certain hardware and software resources. Although the IBM PC has the necessary hardware, MS-DOS—the operating system of the masses—currently lacks any support for multitasking. A new DOS with the requisite features has been promised by IBM and Microsoft "real soon now."

Fortunately, you're not completely abandoned when it comes to software support. Some of the more recent computer languages such as Ada and Modula-2 inherently support multitasking. Even earlier languages such as Pascal and C have the necessary resources if you're willing to write a few routines. In this article I describe how to add multitasking to standard Turbo Pascal. Specifically, I illustrate how to add a cooperative, round-robin task scheduler (kernel) to any Turbo Pascal program using only a few short procedures. The approach I take allows cookbook use of the multitasking routines—that is, you don't have to understand fully how the procedures work to use them productively. Further, you won't need to write any assembly-language code; all the code can be written in high-level Pascal.

After describing the basics of the implementation, I cover the data structures used for interprocess communication and synchronization, hardware interrupt servicing in the multitasking context, and finally an example that illustrates how they all fit together. Please note, the techniques presented in this article apply only to the MS-DOS implementation of Turbo Pascal. A CP/M or Macintosh version would require major modification.

This multitasking kernel was developed as part of the design of a PC-based serial protocol analyzer. The software allows asynchronous acquisition and display of serial data in two directions simultaneously. The serial protocol analyzer application illustrates both of the classical reasons for using multitasking because it provides a minimum response time to external stimuli (serial data) and a natural separation of largely unrelated processes—that is, the acquisition of serial data and management of the computer display.

Implementation

Before delving into the details of the implementation, it's necessary to understand the properties of the multitasking kernel presented here.

The kernel utilizes a cooperative as opposed to a preemptive task-switching mechanism. That is, each task that runs voluntarily gives up control of the CPU so that other tasks can run. This is quite different from the time-slicing preemptive algorithm used in many multitasking schemes. (The term *multitasking* originally implied time slicing.) In the MS-DOS/Turbo Pascal environment, the use of cooperative task switching is beneficial in two ways: first, it solves the reentrancy problem inherent in MS-DOS, and second, it precludes saving the CPU registers during a task switch as Turbo Pascal does not expect CPU registers to be preserved between procedure invocations.

This multitasking kernel uses a round-robin, equal-priority task scheduler. The scheduler moves from one task to the next ready task without regard for task priority. If a task is ready, it will run. Task priority could easily be added to the kernel, but the application for which the kernel was designed did not require it.

Each task utilizes two data structures for its operation. The first is the task control block, or TCB. Figure 1, page 43, shows the layout of a task control block. Example 1, page 43, shows the equivalent Turbo Pascal code. *Bptr* is a pointer into the stack segment of the *BP* register storage location in this task's stack frame. *Bptr* is declared an integer instead of a pointer because the segment into which it points is already known to be the stack segment. *Link* is a pointer to the next *TCB* in a circularly linked list of *TCBs*. *State* indicates the current state of this task with:

Craig A. Lindley, 6 Sutherland Pl., Manitou Springs, CO 80829. Craig works for ROLM Corp. as a software engineer involved in real-time telephony control.

0 = ready to run
1 = waiting for a signal to run
2 = running

Id is an identifying number assigned to this task when it was created. It is currently used only in debugging of a multitasking application.

All task control blocks are located in Turbo Pascal's heap area. The second kernel data structure is the stack frame. As you might expect, the stack frame for each task is located in the 808x stack segment. The stack frame is the stack area for a single task. Each task has its own unique stack frame the size of which is determined when a task is first executed. Many different types of information can be stored in a task's stack frame including:

- procedure return addresses
- parameters passed to procedures
- local variables
- CPU registers during the servicing of hardware interrupts

Figure 2, right, shows how the data structures relate in a program that has three tasks competing for the use of the CPU. In this snapshot, task 1 is currently running. This is indicated by its state being set to 2 and the current pointer, or *CP*, pointing at its task control block. When and if task 1 relinquishes control of the CPU, task 2 will then begin running as it is the next ready task in the linked list. Notice how the operation of this multitasking kernel spans all three of the 808x CPU segments.

Multitasking Kernel Routines

Five Turbo Pascal procedures form the basis of the multitasking kernel. They are *Fork*, *Yield*, *Wait*, *Send*, and *Pause*. Each of these routines manipulates the structure shown in Figure 2. The multitasking kernel's code is shown in Listing One, page 52.

Fork is used to create, or spawn, another task for the CPU to execute. It is modeled closely on the Unix procedure of the same name. *Fork* sets a global variable called *child_process* to indicate whether or not the fork operation was successful. This variable is used to modify program flow depending on the status of the fork operation. If *child_process* is returned true, the fork was successful. It is important to realize that, when fork is successfully executed, control is not returned to the parent task but to the new task just forked—the child process. The parent task is suspended (the task state is set to ready) until given another chance to run. A look at Example 2, page 44, will help clarify the operation of the *Fork* procedure.

On return from the *Fork* procedure, *Newtask* is started in an environment that is different from the environment of the main program. The structure of all tasks is the same, as is discussed later. When *Newtask* gives up control of the processor via a yield or wait, you revert to the main program's environment via a task switch and reexecute the conditional statement shown above. This time, however, the *child_process* variable will be false (reset by the *Newtask* code), so the invocation of

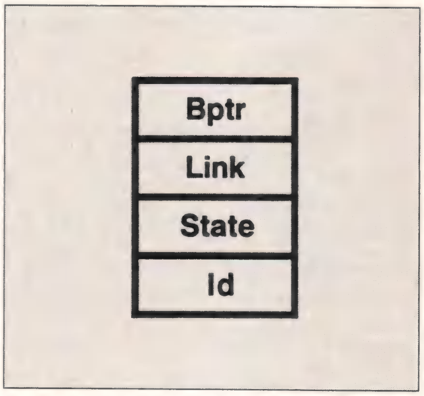


Figure 1: *Task control block*

```
tcbptr = ^ tcb      {a pointer to a TCB}

tcb = RECORD
 Bptr:  integer;  {Bptr storage}
 Link:  tcbptr;   {Link storage}
 State: byte;     {Current state}
 Id:    byte;     {Task Id}
End;
```

Example 1: *Equivalent Turbo Pascal code for Figure 1*

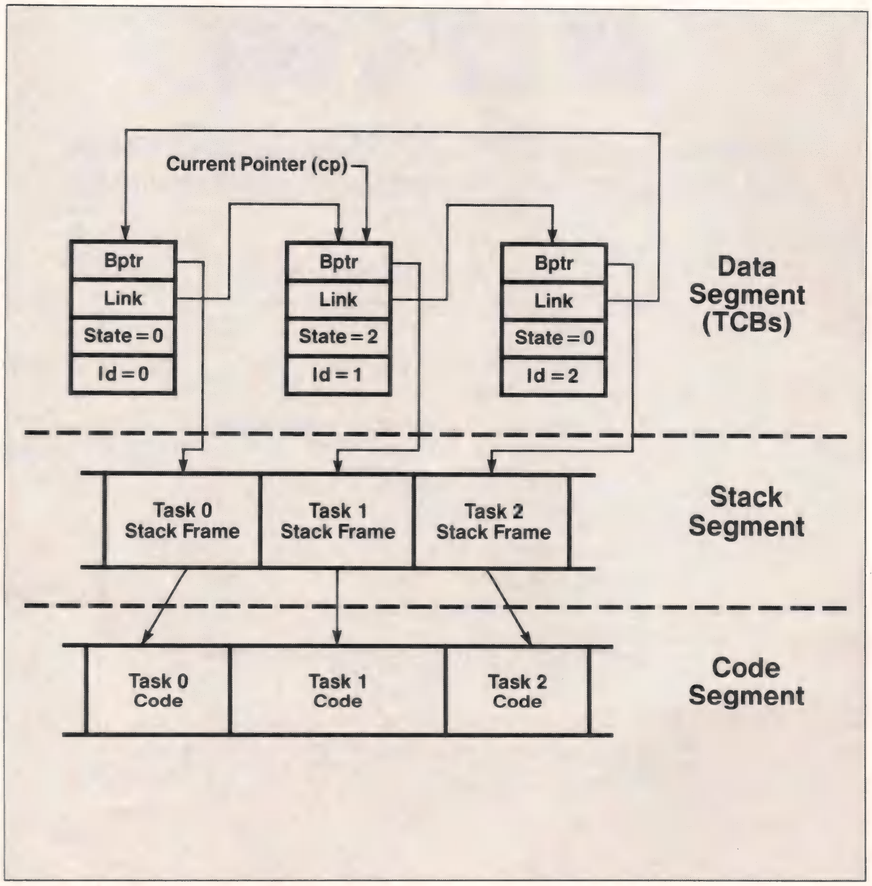


Figure 2: *Program snapshot*

(continued from page 43)

task 1 is skipped and the execution of the main program code continues.

The processor can execute only one program at a time, so while the main program code is executing, the newly forked *Newtask* routine is quiescent. If the main program yielded, however, its execution would pause while *Newtask* continued to run. In this multitasking kernel you can fork as many tasks as the stack segment has stack space available for and the data segment has TCB space available for.

```
.
.the main program code is running here
.
Fork;                          {fork a new task}
If child_process = true then   {if fork was successful}
  Newtask;                     {start new task running}
```

Example 2: Fork *procedure, which manipulates structure in Figure 2*

To help you understand the kernel's operation, here is a detailed breakdown of *Fork*'s operation:

1. It checks to see if there is enough stack space available for a new task. The size of the allocated stack is determined by the global variable *Task_stack_size*, which can be changed between calls to *Fork* if the stack requirements change. Always allocate more stack area than you think you'll need because, if a task crosses its stack boundary, your program will crash and burn. It is advisable to allocate at least 128 bytes of stack more than your program will require to allow for MS-DOS calls and hardware interrupts.
2. It saves the *Bptr* of the currrently executing task in the current TCB so that it can be restored when this task runs again.
3. It calls the Pascal procedure *New* to allocate space for the new task's TCB in Turbo Pascal's heap.
4. It links the new TCB into the linked list of TCBs.
5. It sets the state of the new task's TCB to running as it will be upon return from *Fork*.
6. It points *CP* at this new TCB.
7. It gets the next task's ID number and stores it in the new TCB.
8. A portion of the old task's stack contents are then copied into the new stack frame. This allows return from *Fork* into the new task's environment.
9. It updates the variable *frame_ptr* to reflect the hunk of stack area just allocated to this new task.
10. It sets the *child_process* variable to true.

When all these operations have been completed, the return from *Fork* starts the new task running in its own, newly created environment.

Yield is used by the current task to give up control of the CPU voluntarily to the next ready task. If only a single task is running, *Yield* cannot do anything except output an error message and halt because a programming error has been made. When a task that has yielded runs again, it will continue its execution at the Pascal statement following the yield. It is very important for a task to yield periodically because otherwise it will gain

control of the CPU and might never relinquish it. Under these conditions none of the other tasks would be given any time to run. *Yield* statements should be placed liberally in a task's code, especially in loops that might take quite a while to complete.

The format of all tasks should resemble that shown in Example 3, right.

In other words, every task should be in the form of an infinite loop that never returns to the main program but that does yield to the other tasks.

The actions performed when *Yield* is executed are:

1. The *Bptr* into the current task's stack frame is saved in the current task's TCB.
2. The state in the TCB is changed from running to ready.
3. The linked list of TCBs is traversed until the next ready task is located. *CP* is then made to point at this new TCB.
4. The *Bptr* for the task to run is retrieved from its TCB and stored in the 808x processor's *BP* register, as required by Turbo Pascal. When return from *Yield* is performed, the environment is changed to that of the next task and execution continues from where that task previously yielded. It is quite possible, for example, for task 3 to yield and for task 15 to begin execution if task 15 is the next ready task in the linked list.

Wait is used by a task to voluntarily suspend its execution indefinitely until some external stimulus is applied. The external stimulus is a *Send* operation, which I describe next. Note that a waiting task does not consume any CPU time—it is not polling for the external stimulus to be applied (which would require CPU time) but is completely dormant. As far as the CPU is concerned, a waiting task is nonexistent.

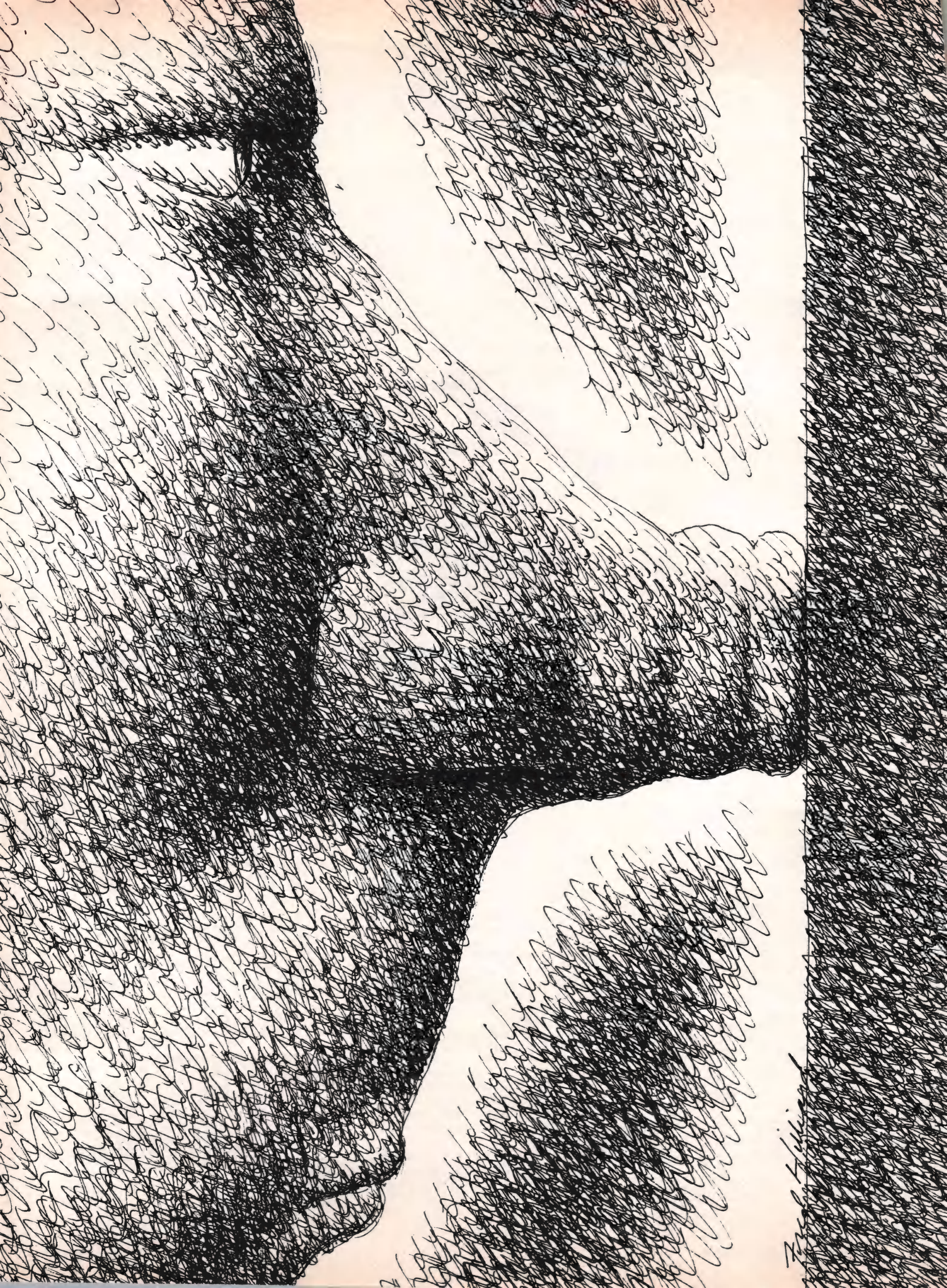
The code and therefore the operation of *Wait* is similar to that of *Yield*. The differences are:

1. The state of the current task is changed from running to waiting instead of to ready.
2. *Wait* requires a parameter to indicate what stimulus the task is to wait for. The parameter is a pointer to a task control block pointer (*tcbptr*), which is stored in the variable *waitfor*. *Waitfor* must be initialized before the *Wait* procedure call is executed.

Send is used to wake up a waiting task. It changes the state of the task being signaled from waiting to ready so that it will run the next time the processor gets to it.

```
Procedure Newtask;
Var
 any required local variables
Begin
 child_process := false;  {resets the global variable}
 Repeat
  New task code goes here
  Yield;
 Until False;
End
```

Example 3: *Task format*



(continued from page 45)

Pause is used to suspend a task's execution for a specified number of one-quarter-second intervals, or ticks. The tick count is a signed integer value with a maximum count of 32,767, representing a maximum delay of approximately 2.5 hours. This implementation of *Pause* is crude but effective. It relies on the fact that all tasks yield periodically, so *Pause* can check to see if the tick count has been satisfied and therefore whether the paused task is to be made ready to run.

Intertask Communication and Synchronization

Because of the asynchronous nature of tasks running in the multitasking environment, data sharing requires special considerations. Two of the more important problems that must be overcome to allow a multitasking system to be viable are:

1. Data passed back and forth between tasks must be absorbed and buffered. This is necessary to equalize the different rates at which tasks produce and consume data.
2. Important data areas or system resources must be protected from use by more than one task at a time.

The classical method used to solve the first problem is with a data structure known as the first in, first out (FIFO) list, or queue. The second problem can be solved using a data structure called a semaphore. Both of these structures, along with the appropriate support routines, are implemented in this kernel.

FIFOs

The implementation of FIFOs in this multitasking kernel is very generalized. By this I mean that the same general techniques can be used to maintain a queue without regard for the type of data stored. A simple FIFO using bytes is shown in the example program of Listing One. FIFOs for storage of other types of data—including real numbers, strings, or records—could just as easily be implemented using the general techniques presented here.

In the example program, a byte FIFO is used to buffer intertask data. To use the generalized FIFO routines, you must perform the following steps:

1. Generate a record structure describing the type of data you want to store in the FIFO. The structure for the byte FIFO is:

```
bytefifo = RECORD
  ovd: overhead; {another record
                           used to}
                  {manage the fifo
                              data}
  data: ARRAY[1 . . bytefifosize] OF
                              BYTE;
End;
```

It is important to use the same overhead record regardless of the type of data stored in the data array. This allows the same generalized techniques to be used to manage the queue.

2. Declare an instance of the FIFO:

```
inbuffer: bytefifo;
```

3. Initialize the FIFO overhead data structure as follows:

```
Initialize_fifo(inbuffer.ovd);
```

This sets the fields in the overhead record to indicate an empty FIFO. The fields as initialized are shown in Table 1, right.

4. The final step is to write routines similiar to *put_byte* and *get_byte*, which can store/retrieve items of data from the FIFO.

Notice the use of the kernel wait and signal routines in the *get_byte* and *put_byte* procedures. Putting the *Not_empty* and *Not_full* signals into the overhead record for all FIFOs makes use of the FIFO routines extremely convenient in this multitasking environment. If, for example, your program calls *put_byte* to store a byte of data in the *inbuffer* FIFO and *inbuffer* is currently full, your task will automatically be put to sleep until a byte is removed from *inbuffer*. As soon as there is room in *inbuffer*, *put_Byte* will store the byte in *inbuffer* and return to your program. Conversely, if a task is waiting for data to be placed in *inbuffer*, it will automatically awaken as soon as *put_byte* places data into the previously empty FIFO.

Semaphores

Semaphores are generally used as a tool for synchronization in a multitasking environment. They can be used to initiate a synchronized action or to provide mutual exclusion for a system resource. Three routines are provided in the kernel for use with semaphores: *Initialize_semaphore*, which initializes the semaphore data structure; *Alloc*, which is used to claim a resource for a task's private use; and *Dealloc*, which releases a resource from a task's control. Many other semaphore manipulation procedures are possible but have yet to be required in my serial protocol analyzer application.

Semaphore usage requires the following steps: first, an instance of a semaphore must be declared:

```
printer_lock: semaphore;
```

and second, the semaphore must be initialized:

```
Initialize_semaphore(printer_lock);
```

As an example of how the semaphore procedures *Alloc* and *Dealloc* could be used, consider the use of a printer in a multitasking environment. Suppose many tasks needed to produce printed output on a printer. If some form of printer control were not imposed on which tasks have access to the printer, the resulting printed output could become a jumbled mess of intermingled charac-

`Count`	= 0	(number of items in the FIFO is 0)
`Inptr`	= 1	(array index of where items placed in the FIFO will be stored)
`Outptr`	= 1	(array index of where items removed from the FIFO will be obtained)
`Not_empty`	= `Nil`	(specialized field, discussed later)
`Not_full`	= `Nil`	(specialized field, discussed later)

Table 1: *Fields in overhead register as initialized*

How to Work with MS-DOS and Make It Work For You!

Program Interfacing to MS-DOS

by William Wong

Originally featured in *Micro/Systems Journal* program, **Interfacing to MS-DOS** provides ten concise articles that will orient any experienced programmer to the MS-DOS environment. All source code discussed is also contained on disk.

Topics include: program construction, character base input and output functions, and file access. You'll also find a discussion of CP/M style vs. Unix-style DOS file access, sample program files, and a detailed description of how to build device drivers. A device driver for a memory disk and a character device driver are provided on disk with full source code.

Program Interfacing to MS-DOS	#Item 166	$29.95

Taming MS-DOS

by Thom Hogan

Learn how to make DOS work for you! Taming MS-DOS takes you beyond the basics, picking up where your DOS manual leaves off.

- Learn to maximize your batch files with routines using redirection, filters and pipes. You'll find routines that prevent accidental reformatting of your hard disk, redefine function keys and locate files within subdirectories. You'll learn to implement a DOS help system with help text files, a menu system that interprets keyboard input, and a routine for quick redefinition of function keys.
- Learn to customize CONFIG.SYS to maximize the performance of your system and how to use ANSI.SYS to tailor your system prompt and monitor attributes to fit your needs.
- Taming MS-DOS includes nearly 50 ready-to-use programs that increase DOS's functionality. Now you can easily rename directories and disk volumes, change file attributes, check available RAM and disk memory, display a memory resident clock, and assign DOS commands to ALT keys.
- Quick reference charts provide easy access to batch command syntax, CONFIG.SYS syntax and ANSI.SYS command strings.

All programs, including batch files and DOS enhancements, are available on disk with full source code.

Taming MS-DOS	**Item #060**	**$19.95**
Taming MS-DOS with disk	**Item #061**	**$34.95**

Return form OR

In CA call 800-356-2002

YES! Please send me:

Item #166 Program Interfacing to MS-DOS	$29.95	______
Item #060 Taming MS-DOS	$19.95	______
Item #061 Taming MS-DOS with disk	$34.95	______
Item #090 SK: The System Kernel	$49.95	______
Item #091 DS: The Display Driver	$39.95	______
Item #092 FS: The File System	$39.95	______
Item #70-4 Demo disk	$5.00	______
Subtotal		______
CA Residents add tax		____%____
Add $2.25 per item for shipping		______
TOTAL		______

☐ Check Enclosed. **Make Payable to M&T Publishing, Inc.**

Charge my ☐ VISA ☐ M/C ☐ Amer. Exp

Card # ______________________ Exp. ______

Name ______________________

Address ______________________

City ______________________ State______ Zip ______

ters. To prevent this, you could declare a semaphore (*printer_lock*, for example) for use with the *Alloc* and *Dealloc* procedures. Each task that required access to the printer would first *Alloc* the printer semaphore before attempting to print and would *Dealloc* the printer when finished. This would prevent the intermixing of printed output as each task would have exclusive access to the printer as long as necessary to complete its output. Another task awaiting the use of the printer would be put to sleep until the printer was available.

The same techniques used to control access to a system device such as a printer can also be used to control access to shared memory areas and system data structures. By using *Alloc* and *Dealloc* in code that modifies an important data structure, you can be assured the complete structure will be modified (even though the task doing the modification repeatedly yields) before access is granted to another task.

Interrupt Processing

Any real-time multitasking system must, by its very nature, provide a close coupling between real-time interrupts and the execution of tasks. Interrupts, like tasks, are another method for dealing with the asynchronous nature of real-world events. Whatever the source of the interrupt, there should be a uniform, well-defined method for handling it in a multitasking environment, and the method chosen should hopefully be transparent to the programmer.

In keeping with this philosophy, interrupt handling within the multitasking environment presented here is performed in Turbo Pascal instead of in assembly language. This is made possible by the use of two small in-line code procedures that must bracket the procedure call to your Turbo Pascal interrupt service routine. The first in-line routine, called the interrupt service routine preamble, contains the 808x code shown in Example 4, above.

The function of this code is to save all the registers used by Turbo Pascal in servicing your interrupt, to establish access to the data area used by your program, and finally to reenable the interrupts.

Turbodseg must be initialized with the data segment value used by Turbo Pascal for storing the data created in your program. This value is required by the interrupt service routine to locate and have access to all the data in your program. *Turbodseg* is actually a typed constant rather than a variable. I used a typed constant because it is stored in the code segment instead of the data segment portion of memory. If a variable were used, it would be stored in the very area of memory—the data segment—you are trying to locate. Storing *Turbodseg* in the code segment

```
Push AX,BX,CX,DX,DI,SI,ES,DS {save all regs modified by}
                             {the interrupt routine}
mov  AX,CS:turbodseg         {get the segment that contains}
mov  DS,AX                   {turbo's data. Make it the dseg}
Sti                          {turn interrupts back on}
```

Example 4: *Preamble to interrupt service*

(continued from page 49)

allows it to be accessed when an interrupt occurs because the interrupt procedure is contained in the same code segment as the rest of your program. Once an interrupt occurs, the value stored in *Turbodseg* is moved into the processor's *DS* register, allowing your interrupt service routine access to all your program's data.

After the preamble code has run, control is passed to your interrupt service routine. Your code can perform any function you desire as long as you keep a few facts in mind. First, the processor registers and flags have been saved onto the stack of an interrupted task, so be sure you don't nest procedures or create local data to such an extent that the stack overflows. Second, do not use any DOS functions (for example, *int 21h*) as they are not reentrant; on the other hand, you can use BIOS functions.

It is a good programming practice to keep interrupt routines as small as is practical. This rule is especially true in the context of multitasking. If large amounts of processing need to be performed, it might be wiser to signal a waiting task to do the required work. How interrupt service routines are written, however, is largely mandated by the type of processing that must be accomplished.

After your Turbo Pascal interrupt service routine terminates, the interrupt postamble code must execute. It performs the inverse function of the preamble routine by restoring all the processor registers to their pre-interrupt state and then doing all the miscellaneous housecleaning necessary to clean up after the interrupt code. The postamble routine code is shown in Example 5, below.

The structure of your interrupt service routine should then be similar to the following:

```
Procedure
your_interrupt_service_routine;

Begin
  preamble code;
  interrupt_service_routine;
  postamble code;
End;
```

It is very important that the interrupt service routine *(your_interrupt Service_routine* in the structure shown above) does not allocate any local data. This is necessary because the preamble and postamble routines are not smart enough to manage the stack correctly when locals are present. The actual interrupt procedure *(interrupt_service_routine)* can use as much local and global data as it requires.

The kernel can be used in a cookbook fashion in which all code can be written in high level Turbo Pascal.

After you have written your interrupt service routine, you must install it in the PC environment before you can execute it. Two methods exist for installing an interrupt routine: you can either store the address offset and code segment values for the routine in the processor's interrupt table directly, or you can call MS-DOS to do the installation for you. The second method is perferred, and you should use it whenever you install interrupt service routines. The example program shown in Listing One shows an interrupt-driven serial input routine written in Turbo Pascal along with the MS-DOS code necessary to install it.

```
Cli                               {interrupts off for a second}
Pop  DS,ES,SI,DI,DX,CX,BX,AX      {restore the registers}
Pop  BP                           {throw away the Sp value}
Pop  BP                           {restore the Bp register}
Iret                              {return from interrupt}
```

Example 5: *Postamble to interrupt service*

The Example Program

I have used a dumb terminal program throughout this article to illustrate some of the multitasking kernel's features. The complete program is shown in Listing Two, page 62. Listing Three, page 70, includes all the RS-232 support routines necessary to compile and run the example program.

The example program illustrates the following concepts:

- creation of four concurrent tasks
- the use of FIFOs to pass data between tasks
- processing of serial interrupts in Turbo Pascal

This program can be used as a template for the generation of your multitasking programs.

Conclusions

In this article I've presented a multitasking kernel for use with MS-DOS and Turbo Pascal. Included are many functions useful either for a real-time control application or just for experimenting with the concepts presented. From a programmer's perspective, the kernel can be used in a cookbook fashion in which all code (for both the tasks and the interrupt service routines) can be written entirely in high-level Turbo Pascal. Feel free to incorporate these concepts and code in your own programs. I'd appreciate hearing from you if you come up with any novel uses or extensions for the kernel.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 216. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 52.)

Listing One *(Text begins on page 42.)*

```
{$K-}              {Compiler switch - never change}

{*************************************************}
{***              Listing One                 ***}
{***              Turbo Pascal                ***}
{***          Multitasking Kernel             ***}
{***              written by                  ***}
{***            Craig A. Lindley              ***}
{***                                          ***}
{***    Ver: 1.3     Last update: 03/11/87    ***}
{***                                          ***}
{*************************************************}

CONST

   task_stack_size  = 256; {stack size for each}
                           {task}
   turbodseg: integer = 0; {storage for turbos}
                           {data segment value}


TYPE

{possible states for a task}
   task_state = (ready,waiting,running);

{808X register set}
   register_type = RECORD
   CASE integer OF
      1: (ax,bx,cx,dx,bp,si,di,ds,es,flags:integer);
      2: (al,ah,bl,bh,cl,ch,dl,dh         :byte);
   END;


{Task control block (tcb) structure}

 tcbptr = ^ tcb;        {ptr to tcb}

 tcb = RECORD
    link:  tcbptr;      {link to next tcb in dseg}
    bptr:  integer;     {base ptr offset in sseg}
    state: task_state;  {ready, waiting, running}
    id:    byte;        {task number}
 END;

 waitptr = ^tcbptr;     {ptr to ptr to tcb}
                        {used for passing parms}
                        {to wait}

{This fifo overhead structure is the same for}
{all fifo types regardless of the items to be}
{stored in the fifo. The byte fifo is an example}
{of just one possible type of fifo.}

 overhead = RECORD      {fifo overhead data}
                        {structure}
    count,              {# of items in fifo}
    inptr,              {ptr to where items are}
                        {stored}
    outptr:   integer;  {ptr to where items are}
                        {fetched}
    not_empty,          {ptrs to waiting tasks}
    not_full: tcbptr;
 END;


 bytefifo = RECORD      {definition of a byte fifo}
    ovd:  overhead;     {fifo overhead}
    data: ARRAY[1..bytefifosize]
          OF byte;      {byte fifo data area}
 END;


 semaphore = RECORD     {Semaphore data type}
    count:  integer;    {number of times signaled}
    signal: tcbptr;     {pointer to waiting task}
                        {if there is one}
 END;
```

(continued on page 54)

Listing One *(Listing continued, text begins on page 42.)*

```
{******** Begin Multitasking Variables *********}

VAR

   cp,                     {current task pointer}
   new_tcb_ptr,            {ptr to new tcb in dseg}
   temp_ptr:   tcbptr;

   waitfor:    waitptr; {address of item to}
                        {wait on}
   stk,bp:     integer; {variables for setting}
                        {808X sp and bp}
   frame_ptr:  integer; {stack frame pointer}
   next_id:    integer; {next task id number}
   i:          integer;
   child_process: boolean; {fork successful flag}


{******** Begin Multitasking Procedures ********}


PROCEDURE Fork;         {fork off a new task}

{This procedure manipulates Turbo Pascal's stack}
{frame as required to fool it into operating in}
{a new task's environment.}

BEGIN

   child_process:=false;    {indicate the parent}
                            {process until proven}
                            {otherwise}
   {check if enough stack space for a new task}

   IF abs(frame_ptr - task_stack_size) > 0 THEN
   BEGIN                    {if enough}
      INLINE($89/$26/stk); {get 808X Sp to}
                            {calculate Bp pointer}
      cp^.bptr:=stk+2;      {save Bp ptr in this}
                            {frame}
      new(new_tcb_ptr);     {allociate new tcb}

      {link new tcb into scheduler loop}
      {make its state running and give it an id}

      new_tcb_ptr^.link:=cp^.link;
      cp^.link:=new_tcb_ptr;
      new_tcb_ptr^.state:=running;
      next_id:=next_id+1;
      new_tcb_ptr^.id:=next_id;

      cp^.state:=ready;     {old frame is ready}

      {copy old stack to new stack frame}
      FOR i:=0 TO 5 DO
         mem[sseg:frame_ptr-6+i]:=mem[sseg:stk+i];

      {make Bp storage in stack frame point at}
      {this frame}

      memw[sseg:frame_ptr-4]:=frame_ptr;
      bp:=frame_ptr-4;      {calculate Bp pointer}

      INLINE($8B/$2E/bp);   {set 808X Bp reg to}
                            {this new value}

    {reserve stack frame space}
    frame_ptr:=frame_ptr-task_stack_size;
    cp:=new_tcb_ptr;        {cp points at new task}
    child_process:=true;    {indicate child process}
  END;

END;



PROCEDURE Yield;

{This procedure cause the executing task to}
{relinquish control of the CPU to the next ready}
{task.}
```

```
BEGIN

   child_process:=false;    {reset variable}
   IF cp^.link <> cp THEN   {must have more than}
                            {one task forked to be}
                            {able to yield}
   BEGIN
      INLINE($89/$26/stk);  {get 808X sp}
      cp^.bptr:=stk+2;      {save Bp ptr in}
                            {current task frame}
      cp^.state:=ready;     {yielded task ready}
      temp_ptr:=cp;

      {look for next ready task in scheduler loop}
      {there must be at least one or else}

      WHILE (temp_ptr^.link^.state <> ready) DO
         temp_ptr:=temp_ptr^.link;

      cp:=temp_ptr^.link;   {cp points at new task}
      cp^.state:=running;   {indicate running}
      bp:=cp^.bptr;         {get the bp of task}

     INLINE($8B/$2E/bp);    {restore it to 808X bp}
   END
   ELSE
   BEGIN
      writeln('Cannot yield only single task running');
      halt;
   END;

END;



PROCEDURE Wait;  {put current task in wait mode}
                 {until a send makes it ready}

{Due to constraints of this kernel, parameters}
{cannot be passed directly to the wait procedure.}
{To overcome this limitation, a global variable}
{called waitfor is used. The address of the}
{tcbptr on which to wait should be stored in}
{waitfor. See the fifo routines for an example of}
{the proper usage of Wait.}

BEGIN

   child_process:=false;    {reset variable}
   IF cp^.link <> cp THEN   {must have more than}
                            {one task forked to be}
                            {able to wait}
   BEGIN
      waitfor^ := cp;       {waitfor points at the}
                            {current task}

      INLINE($89/$26/stk);  {get 808X sp}
      cp^.bptr:=stk+2;      {save it in current}
                            {task frame}
      cp^.state:=waiting;   {task now waiting}
      temp_ptr:=cp;

      {look for next ready task in scheduler loop}
      {there must be at least one or else}

      WHILE (temp_ptr^.link^.state <> ready) DO
         temp_ptr:=temp_ptr^.link;

      cp:=temp_ptr^.link;   {cp points at new task}
      cp^.state:=running;   {indicate running}
      bp:=cp^.bptr;         {get bp for this task}
      INLINE($8B/$2E/bp);   {restore it to 808X bp}
   END
   ELSE
   BEGIN
      writeln('Cannot wait only single task running');
      halt;
   END;

END;
```

(continued on page 57)

Listing One *(Listing continued, text begins on page 42.)*

```
PROCEDURE Send(VAR s:tcbptr);

{Make the specified task ready for next scheduler}
{go around}

BEGIN

   s^.state:=ready;      {task state is ready}
   s:=NIL;               {clear pointer}

END;



PROCEDURE Pause(t:integer);

{Pause the execution of a task for t 1/4 sec}
{intervals. Note even t results in more}
{accurate timmings.}

 FUNCTION tic_count : integer;

 {Get the current tic count from the Bios}

 VAR

    regs: register_type;

 BEGIN

    regs.ax:=0;           {request clock tic read}
    intr($1A,regs);
    tic_count:=regs.dx;  {LSB of count in dx}

 END;


VAR

   tics,i: integer;

BEGIN

   tics:=0;              {initial tic count to 0}
   IF t > 0 THEN         {if a legal tic count}
   BEGIN
      FOR i:=1 TO t DO   {250 msec = 4.55 tics}
         IF odd(i) THEN {use this algorithm for}
                        {approximation}
            tics:=tics+4 {250 msec = 4.5 tics}
         ELSE
            tics:=tics+5;

      {add tics to current tic_count to get}
      tics:=tics+tic_count;    {target time}

      REPEAT
         yield;    {return when tic count is}
                   {reached or exceeded}
      UNTIL tics <= tic_count;
   END
   ELSE
      writeln('Bad tic count specified');

END;


PROCEDURE Init_Kernel;

{This procedure initializes the multitasking}
{for use. It sets up the TCB for task 0 and}
{indicates that it is running.}

Begin

   turbodseg:=dseg;      {save turbo data segment}
   frame_ptr:= $FFFE;    {initial stack location}
   next_id:=0;           {first task id}
   new(new_tcb_ptr);     {get new tcb in dseg}
```

(continued on next page)

Listing One *(Listing continued, text begins on page 42.)*

```
   cp:=new_tcb_ptr;        {cp points at tcb}
   cp^.link:=cp;           {points at itself}
   cp^.state:=running;     {in running state}
   cp^.id:=next_id;        {id = 0}

   {now allocate 1st frame for task 0}
   frame_ptr:=frame_ptr-task_stack_size;

End;


{************ Begin FIFO Procedures ************}


PROCEDURE Initialize_fifo(VAR o:overhead);

{Initialize a fifo's overhead data structure.}
{This procedure will work with any type fifo.}
{This makes the fifo appear empty.}

BEGIN

   o.count:= 0;            {count is empty}
   o.inptr:=1;             {ptrs to 1st entry}
   o.outptr:=1;            {put in and take out at}
                           {entry 1}
   o.not_empty:=NIL;       {signals to nil}
   o.not_full:=NIL;

END;


PROCEDURE Put_byte(b:byte; VAR f:bytefifo);

{This procedure manages the input of data into}
{a byte fifo. If the fifo is full when this}
{procedure is called, the task that called it}
{will be put to sleep automatically until there}
{is room in the fifo for the data byte.}
{The fifo overhead data structure is modified}
{whenever a byte is placed into the fifo}

BEGIN

   WITH f.ovd DO
   BEGIN                   {check if fifo full}
      IF count = bytefifosize THEN
      BEGIN                {if so go to sleep}
         waitfor := addr (not_full);
         wait;
      END;                 {when not full add}
      count:=count+1;      {one more to count}
      f.data[inptr]:=b;    {store the byte}
      inptr:=inptr+1;      {bump input pointer}
      IF inptr > bytefifosize THEN
         inptr:=1;         {wrap ptr if necessary}

      {if waiters for this fifo wake them}

      IF not_empty <> NIL THEN
         send(not_empty);
   END;

END;


FUNCTION Get_byte(VAR f:bytefifo) : byte;

{This procedure manages the output of data from}
{a byte fifo. If the fifo is empty when this}
{procedure is called, the task that called it}
{will be put to sleep automatically until there}
{is data in the fifo to retrieve.}
{The fifo overhead data structure is modified}
{whenever a byte is removed from the fifo}

BEGIN

   WITH f.ovd DO
   BEGIN                   {check if fifo empty}
      IF count = 0 THEN
```

(continued on page 60)

Professional Programming Products

by BCSoft

FREE

PC-WRITE™ text editor
with every purchase.

PROGRAMMERS AND SOFTWARE DEVELOPERS - LOOK AT THESE PRODUCTS!

NEW

Quick-Tools™
for QuickBASIC™ users!

- A comprehensive library of over 90 subroutines and functions directly CALLable from your QuickBASIC programs.
- SORT routines allow sorting of single and multidimensional arrays.
- Binary search functions.
- Unique screen handling functions allow splitting screens, fast scrolling, special string printing, character attribute control, phantom cursors, etc.
- Keyboard scanning and status calls, with field inputing and decoding. On screen editing of input fields.
- Unique file handling routines.
- FREE Updates and phone support!
- Plus much, much more!

Only $129.95 Complete

NEW

NET-TOOLS™
NETBIOS Programming Tools

- NET-TOOLS allows you to write programs for ANY NETBIOS compatible local area network - fast and easily. CALLable from Microsoft Assembler, C, PASCAL, or FORTRAN.
- Add and Delete local names.
- Initiate and Cancel sessions. Just give NET-TOOLS the name of the computer you would like to call, and it will automatically locate the user and make the connection for you.
- Transfer Messages with automatic retries and error detection. Both the datagram and session protocols.
- Redirect Local Devices simply and easily with a single function call.
- Plus much MORE !
- Complete SOURCE CODE is provided, FREE !
- FREE Updates and phone support!

ONLY $149.00 Complete

TURBO.ASM™
For Turbo PASCAL users !

- A unique programming tool which is a must for every Turbo Pascal user. The only package designed for interfacing assembly language with Turbo Pascal.
- Outlines several different ways to add assembly language routines to Turbo programs, some without effecting your code space.
- Fully explains the internal workingsof Turbo Pascal and data passing methods.
- Includes a library of Assembly routines which can be used directly from Turbo Pascal.
- The best way to learn assembly language.
- FREE Updates and phone support!

Only $99.95 Complete

ASMLIB™
The Programmer's Library

- A set of over 210 subroutines to greatly increaseyour productivity. Written in assembly language anddirectly CALLable from Microsoft Assembler, C,PASCAL, and FORTRAN.
- Complete SOURCE CODE provided -FREE!
- Text WINDOWing functions allow up to 64 overlapping windows.
- TERMINATE and STAY RESIDENT programs arewritten easily. Programs can "POP UP" with a simplekeystroke. Use DOS calls from them, also.
- GRAPHICS on the EGA, CGA, and Hercules™ Monochrome.
- Virtual file functions.
- Full FLOATING POINT math and trig with 8087 support.
- Int. driven async. support, plus MUCH MORE!

Only $149.00 Complete.

NO ROYALTIES REQUIRED

To ORDER call or write to:

BC Associates
A division of BCSoft Corporation
3261 N. Harbor Blvd.
Suite B
Fullerton, CA. 92635

VISA, M/C, or COD orders are welcome!

CALL TOLL FREE 1-800-262-8010 in CA. dial 1-714-526-5151

Listing One *(Listing continued, text begins on page 42.)*

```
      BEGIN            {if so go to sleep}
         waitfor := addr (not_empty);
         wait;
      END;
                       {when data is available}
      count:=count-1;  {one less to count}
      get_byte:=f.data[outptr];  {get the byte}
      outptr:=outptr+1;{bump output pointer}
      IF outptr > bytefifosize THEN
         outptr:=1;    {wrap ptr if necessary}

      {if waiters for this fifo wake them}

      IF not_full <> NIL THEN
         send(not_full);
   END;

END;


{********* Begin Semaphore Procedures **********}

PROCEDURE Initialize_semaphore(VAR s:semaphore);

{Initialize a semaphore data structure}

BEGIN

   s.count := 0;         {indicate resource is}
                         {available}
   s.signal:=NIL;        {and that there are no}
                         {waiters}

END;



PROCEDURE Alloc(VAR s:semaphore);

{This procedure allocates exclusive use of a}
{resource to the task that executes it. This}
{claim is maintained even though the task}
{gives up control of the CPU via a yield etc.}

BEGIN

   WHILE s.count <> 0 DO  {wait for semaphore}
                          {controlled resource}
                          {to become available}
   BEGIN
      waitfor := addr (s.signal);
      wait;
   END;                   {then}
   s.count:=1;            {claim it}

END;


PROCEDURE Dealloc(VAR s:semaphore);

{This procedure deallocates a resource.}
{Note this routine yields so the deallocated}
{resource has a chance of being used}
{immediately}

BEGIN

   s.count:=0;       {remove claim on resource}
   send(s.signal);   {and awaken the waiting task}
   yield;            {give other tasks a chance}

END;

{End of kernel procedures}
```

End Listing One

(Listing Two begins on page 62.)

HOT GRAPHICS PACKAGE FOR C PROGRAMS.* $39.95

Everything you need to write dramatic graphics effects into your Eco-C88 C programs. Some of the features include:

- Support for EGA, CGA, and Z100
- Over 100 graphics and support functions, many of which are PLOT-10 compatible.
- Many low level support routines reside outside your small model code-data area
- Can write dots thru the BIOS (for compatibility) or to memory (for speed)
- Graphics function help from CED editor available
- World, pixel or turtle color graphics modes
- 47 standard fill patterns, 17 line dashing patterns, Hershey fonts, plus user defineable fill, dash and fonts
- Supports view areas, rotateable fonts, clipping, arbitrary fill areas, extensive error checking, examples, and user's manual.

A must for the graphics enthusiast and a bargain at only **$39.95**

***Requires Eco-C88 C Compiler.**

NEW POP-UP WINDOWS FOR YOUR C PROGRAMS.

This windowing library allows you to add pop up windows in your C programs quickly and easily. Use them for help windows, selection menus, error messages, special effects—anywhere you need an attention getter. Just some of the features include:

- CGA, EGA, and monochrome support
- Slow mode option for "flicker" displays
- Control any program that goes through the BIOS
- Use up to 255 windows
- No special window commands; use print f ()
- Resize and move windows
- Custom window titles and borders
- Can be used with ANSI device driver
- Most of window's code-data lies outside small model limits
- Use any of the IBM text or block characters
- User's manual and examples

The Windowing Library requires an IBM PC compatible BIOS and the Eco-C88 C compiler.

ONLY $29.95

HANDY LIBRARIAN MAKES LIFE EASIER.

Now you can combine your modules, functions, and subroutines into your own library for easy link commands. Fully compatible with ANY standard OBJ format files (not just Ecosoft's products). With the Ecosoft librarian, you can:

- Add, delete, and extract from a library
- Get table of contents or index of a library
- Combine libraries, control library page size, use switches for combinations, process complex library requests, use wildcards, and do library directives from command files.
- Complete with user's manual

A valuable addition for any programmer.

ONLY $29.95

THE FIRST PROFESSIONAL 'C' COMPILER FOR UNDER $60.

A C compiler with many ANSI enhancements at an unbelievably low price. The Eco-C88 C compiler has:

- Prototyping (the new type-checking enhancement)
- Enum and void data types
- Structure passing and assignment
- All operators and data types
- A standard library with more than 200 functions (many of which are System V compatible for greater code portability)
- CC and mini-make that all but automates the compile process
- 8087 support (we sense the 8087 at runtime — no dual libraries)
- ASM or OBJ output for use with MSDOS linker
- Tiered error messages — enable-disable lint-like error checking
- Fast compiles and executing code
- Expanded user's manual
- CED full-screen program editor

Everything you need at the unbelievable price of $59.95.

Eco-C88 C compiler requires an IBM PC, XT, or AT (or compatible) with 256K of memory, 2 disk drives and MSDOS 2.1 or later.

Orders only:
1-800-952-0472
Technical Information:
(317) 255-6476

NOT COPY PROTECTED.

Ecosoft Inc.
6413 N. College Ave.
Indianapolis, IN 46220

ORDER FORM CLIP & MAIL TO: Ecosoft Inc., 6413 N. College Ave., Indianapolis, IN 46220

ITEM	PRICE	QTY	TOTAL
Flexi-Graph Graphics	$39.95		
Window Library	$29.95		
Eco-Lib Librarian	$29.95		
Eco-C88 C Compiler CED	$59.95		
SHIPPING			
TOTAL (IND. RES. ADD 5% TAX)			

PAYMENT: ☐ **VISA** ☐ **MC** ☐ **AE** ☐ **CHECK**

CARD # ____________ EXPIR DATE ______

NAME ____________

ADDRESS ____________

CITY ____________ STATE ______

ZIP ______ PHONE ____________

CIRCLE 89 ON READER SERVICE CARD

Listing Two

(Text begins on page 42.)

```
PROGRAM Multitasking_Demonstration_Program;

{*************************************************}
{***              Listing Two                  ***}
{***        Multitasking Demonstration         ***}
{***         A dumb terminal program           ***}
{*** utilizing 4 tasks and a serial interrupt ***}
{***            service routine.               ***}
{***                                           ***}
{***              written by                   ***}
{***            Craig A. Lindley               ***}
{***                                           ***}
{***     Ver: 1.0     Last update: 03/11/87    ***}
{***                                           ***}
{*************************************************}

CONST

 bytefifosize = 100;     {max size of byte fifos}


{include the multitasking kernel routines}
{$I multi.pas}

{include the RS-232 functions}
{$I serial.pas}


VAR

   inbuffer,
   outbuffer:     bytefifo;


{***** Serial Interface Support Procedures *****}

PROCEDURE Get_serial_char;

VAR

   b:   byte;

BEGIN

{Get the character from the UART. Place it in}
{inbuffer if there is room, throw it away if}
{not. Signal end of interrupt (EOI level 4 on}
{8259.}

   b := port[portaddress];

   IF inbuffer.ovd.count < bytefifosize THEN
      Put_byte(b,inbuffer);

   port[$20] := $20;

END;



PROCEDURE Serial_Interrupt_Service_Routine;

{This is the new interrupt service routine.}
{It replaces the one MsDos normally uses.}
{See text for details.}

BEGIN

{standard interrupt service routine preamble}
```

```
 INLINE($50/$53/$51/$52/$57/     {Push ax,bx,cx,dx,}
        $56/$06/$1e/             {di,si,es,ds}
        $2e/$a1/turbodseg/       {mov ax,cs:turbodseg}
        $8e/$d8/                 {mov ds,ax}
        $fb);                    {sti}

 Get_serial_char;

 {standard interrupt service routine postamble}

 INLINE($fa/$1f/$07/$5e/$5f/     {interrupts off}
        $5a/$59/$5b/$58/         {Pop ds,es,si,di,}
                                 {dx,cx,bx,ax}
        $5d/$5d/$cf);            {trash sp, restore}
                                 {Bp and iret}

END;


{************ Begin Task Procedures ************}


PROCEDURE Task_0;

{Task 0 gets keyboard input and puts it into}
{outbuffer. If no input available task 0 yields.}
{Note infinite loop structure.}

VAR

   ch:    char;

BEGIN

   writeln('Starting Task 0');
   writeln;
```

```
   REPEAT
      IF NOT keypressed THEN
         Yield
      ELSE
      BEGIN
         read(kbd,ch);
         Put_byte(byte(ch),outbuffer);
      END;
   UNTIL false;

END;


PROCEDURE Task_1;

{Task 1 takes character from outbuffer using}
{Get_Byte and sends them out the serial port.}

BEGIN

   writeln('Starting Task 1');
   REPEAT
      Serialout(Get_byte(outbuffer));
   UNTIL false;

END;


PROCEDURE Task_2;

{Task 2 retrives characters placed in inbuffer}
{by the serial interrupt routine and displays}
{them on the screen. Note:}
{ 1) If no characters are available this routine}
{    yields.
```

(continued on next page)

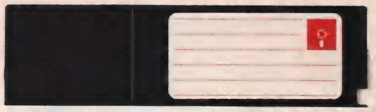
Listing Two *(Listing continued, text begins on page 42.)*

```
{ 2) Interrupts must be disabled while inbuffer}
{    is being accessed. Otherwise the fifo}
{    counter will get confused and this program}
{    will eventually crash.}


BEGIN

   writeln('Starting Task 2');
   REPEAT

      IF inbuffer.ovd.count <> 0 THEN
      BEGIN
         INLINE($FA);    {interrupts off}
         write(chr(Get_byte(inbuffer)));
         INLINE($FB);    {interrupts on }
      END
      ELSE


         Yield;

   UNTIL false;

END;


PROCEDURE Task_3;

{Task 3 monitors and displays the fifo and cursor}
{status. It wakes up every 1/2 second to do so.}
{The cursor position is saved and retrived while}
{the fifo status is being updated on the screen}

VAR

   cursorx,
   cursory: byte;

BEGIN

   writeln('Starting Task 3');
   REPEAT
      pause(2);           {wake every 1/2 sec}
      cursorx := wherex;  {save cursor position}
      cursory := wherey;
      window(1,1,80,25);
      gotoxy(21,25);
      write(cursorx:2);   {write cursor position}
      gotoxy(35,25);
      write(cursory:2);
      gotoxy(58,25);      {write fifo counts}
      write(inbuffer.ovd.count:2);
      gotoxy(72,25);
      write(outbuffer.ovd.count:2);
      window(1,1,80,23);
      gotoxy(cursorx,cursory);
   UNTIL false;

END;


PROCEDURE Initialize_Display;

{This procedure initializes the screen for the}
{demo program. It builds a status line on screen}
{line 25 and then establishes a terminal window}
{so the status line will not be over written.}

BEGIN

   window(1,1,80,25);   {window is full screen}
   CLRSCR;              {clear the screen}

   writeln('Multitasking Demonstration');
   writeln('  A dumb serial terminal program');
   writeln('  Use ^C to abort');
   writeln;
```

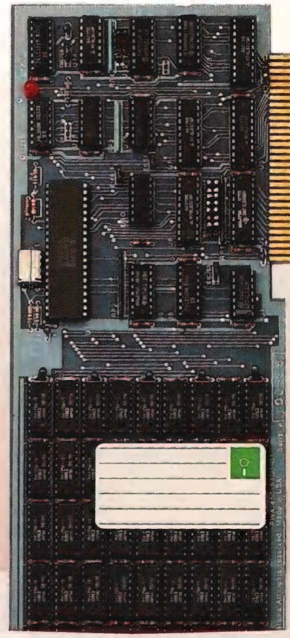
(Listing continued on page 66)

Think fast! Pick the better fit...

5th Year Bonus!
ention this ad when ordering
d get your choice of a V 20-8
eplaces 8088) or $20 off on
your Battery Backup purchase!



FLOPPY DISK.
- Fills time between coffee breaks
- Makes a hard disk seem *fast*
- Your computer appears busy (even if you aren't!)
- Wears out moving parts



SEMIDISK Disk Emulator.
- Gets that job done NOW
- Makes a hard disk seem *slow*
- Maximizes your productivity with anything from databases to compilers
- Totally silent operation

...for YOUR demanding tasks.

SURPRISE! *Neither* is memory mapped, so they don't affect your precious Main Memory. *Both* retain data indefinitely - even with the computer turned off.

THE SEMIDISK SOLUTION. You could invest in a series of "upgrades" that turn out to be expensive band-aids without solving your real problem. Even those "Accelerator" and "Turbo" boards do little to speed up disk-bound computers. If your applications spend too much time reading and writing to disk (and whose don't?), you won't want to settle for anything less than a SemiDisk disk emulator. The SemiDisk comes in 512K and 2Mb capacity. More boards may be added to make up to an 8 Megabyte SemiDrive!

SPEED THAT'S COMPATIBLE. PC, XT or AT, if you need speed, the SemiDisk has it. How fast? Recent benchmarks show the SemiDisk is from 2 to 5 times faster than hard disks, and from 25% faster (writing) to several times faster (random reads) than VDISK and other RAMdisk software that gobble up your main memory.

MEMORY THAT'S STORAGE. Using our small external power supply, with battery backup, your data remains intact through your longest vacation or even a seven-hour power failure!

CELEBRATE WITH US! Now, SemiDisk celebrates its fifth birthday with a special offer for IBM-PC owners. Buy a SemiDisk now and we'll include an 8 MHz V-20 microprocessor (replaces the 8088) to make your new SemiDisk run even faster. Don't need the V-20? We'll take $20 off the price of your Battery Backup Unit!

	512K	2Mbyte
IBM, PC, XT, AT	**$495**	**$ 795**
Epson QX-10	**$495**	**$ 995**
S-100 SemiDisk II	**$795**	**$1295**
S-100 SemiDisk I	**$299**	-------------
TRS-80 II, 12, 16	**$495**	**$ 995**
Battery Backup	**$130**	**$ 130**

**Someday you'll get a SemiDisk.
Until then, you'll just have to...wait.**

SemiDisk

MasterCard VISA

SemiDisk Systems, Inc., 11080 S.W. Allen Blvd., Beaverton, Oregon 97005 (503) 626-3104

CIRCLE 85 ON READER SERVICE CARD

Listing Two *(Listing continued, text begins on page 42.)*

```
    {build the status line}
    gotoxy( 1,25);  write('Status -- Cursor X:');
    gotoxy(25,25);  write('Cursor Y:');
    gotoxy(49,25);  write('Incount:');
    gotoxy(62,25);  write('Outcount:');
    window(1,1,80,23);    {establish terminal window}
    gotoxy(1,8);          {home the cursor}

END;


{************* Begin Main Program **************}

BEGIN {main}

{The main program builds the screen status}
{line, initializes the input and output fifos,}
{initializes the multitasking kernel, installs}
{the serial interrupt handler and then begins}
{forking the individual tasks.}

   Initialize_Display;

   Initialize_fifo(inbuffer.ovd);
   Initialize_fifo(outbuffer.ovd);
   Init_Kernel;

{Initialize and install the serial interrupt}
{handler. Install our interrupt routine in}
{place of the original system IRQ4 handler}

   WITH regs DO
   BEGIN
      ah:=$25;
      al:=$0C;
      ds:=cseg;
      dx:=ofs(Serial_Interrupt_Service_Routine);
      msdos(regs);
   END;

{Set the serial format to 1200 baud}
{1 stop bit, 8 data bits and no parity}

 Setserial(1200,1,8,none);

{Fork off tasks one through three}

 Fork;

 IF child_process THEN
  Task_1;

 Fork;

 IF child_process THEN
   Task_2;

 Fork;

 IF child_process THEN
   Task_3;

{Enable the serial interrupt}

 Enable_serial_int;

{Start Task 0}

 Task_0;

END.
```

End Listing Two

(Listing Three begins on page 70.)

Discover the Difference

We're Programmer's Connection, your best one-stop source for quality programmer's development tools for IBM Personal Computers and compatibles. We invite you to compare us with the other dealers in our industry:

FREE Shipping. Shipping to U.S. customers is FREE via UPS Ground. If you want your order shipped via an express service, we'll only charge you the shipping carrier's standard rate with no special fees. Some dealers charge extra for shipping and then add rush charges for shipping via express services. Others may advertise "free" shipping but make up for it by charging extra handling fees.

Credit Cards. We'll charge your credit card only when we actually ship your order. Some dealers would charge your credit card at the time you place your order. This could leave you waiting for your shipment for weeks or months while they use your money interest-free.

Discounts. We discount all software products — even special order items. Every product in our advertised price list is shown with its list price and discounted price. We want you to know exactly how much you'll save on every product. We don't try to fool you by discounting some products and charging full retail for others.

Consistent Prices. We extend the same current prices to every customer regardless of where they see our ad. Some dealers vary prices in different ads and then ask you to mention which one you saw. This technique allows them to charge you the highest prices possible.

No Hidden Charges. The discount prices you see on the next two pages are all you pay. We don't charge extra for UPS Ground shipping, credit cards, COD orders, purchase orders, sales tax (except Ohio) or special handling (except for non-Canadian international orders).

Guarantees. We offer FREE 30-day no-risk return guarantees and 30-day evaluation periods on most of our products. Some dealers have no return options while others often charge restocking fees of 15% or more.

Quality Products. Our product line consists of hundreds of high quality software development tools specifically for IBM Personal Computers and compatibles. While some dealers try to carry every software product ever written, we carry only those that meet our very high standards for quality and value.

Latest Versions. The products we carry are the latest versions and come with the same manufacturer's technical support as if buying direct. While some dealers may participate in the software gray market, we're authorized to sell every product we carry.

Large Inventory. We have one of the largest inventories of programmer's development products in the industry. Most orders are shipped within 24 hours. And if we don't have a product in stock, we'll get it for you fast.

Meticulous Packaging. We'll give your shipment the extra protection needed to reach you in the best possible condition. First we'll protect your products from moisture by wrapping them in plastic. Then we'll insulate your box with high quality bubble-wrapping instead of the messy styrofoam chips that many other dealers use. Finally, we'll double-tape your box for extra strength.

Independence. Since we're not directly affiliated with any software publisher or developer, we can give you sound, unbiased advice. Unlike some other dealers who have a special interest in promoting only certain products, we'll give you an objective look at the products we carry.

Noncommissioned Staff. Our courteous sales staff is always ready to help you. And if you aren't sure about your needs, our knowledgeable technical staff can give you sound, objective advice. Because they are noncommissioned, you won't be pressured into making a purchase.

As you can see, we're not like the other dealers in this industry. Our customers keep coming back because we consistently provide the highest quality service and the lowest prices. So call us today and discover the difference for yourself.

PROGRAMMER'S CONNECTION
QUALITY · INTEGRITY · Service · SUPPORT · PRICE

ai - expert systems

	List	Ours
1st-CLASS *by Programs in Motion*	495	399
AutoIntelligence *by IntelligenceWare*	990	739
ExpertEDGE Advanced *by Human Edge*	2500	CALL
ExpertEDGE Professional *by Human Edge*	5000	CALL
Experteach II *by IntelligenceWare*	475	339
EXSYS *Development Software by EXSYS*	395	309
EXSYS Runtime System	600	469
Insight 1 *by Level Five Research*	95	75
Insight 2+ *by Level Five Research*	485	379
Intelligence/Compiler *by IntelligenceWare*	990	739
Logic-Line Series 1 *by Thunderstone*	90	85
Logic-Line Series 2 *by Thunderstone*	125	115
Logic-Line Series 3 *by Thunderstone*	150	139

ai - lisp language

	List	Ours
GCLISP *Golden Common LISP by Gold Hill*	495	CALL
GCLISP 286 Developer *by Gold Hill*	1190	CALL
Microsoft LISP *Common LISP*	250	149
QNIAL *Combines LISP & APL by NIAL Systems*	375	339
Starter QNIAL *by NIAL Systems* *New*	99	89
TransLISP *from Solution Systems*	95	79
TransLISP PLUS *from Solution Systems*	195	169

ai - prolog language

	List	Ours
APT *Active Prolog Tutor from Solution Systems*	65	55
Arity Combination Package	1095	979
Expert System Development Pkg	295	229
File Interchange Toolkit	50	44
PROLOG Compiler & Interpreter	650	569
Screen Design Toolkit	50	44
SQL Development Package	295	229
Arity PROLOG Interpreter	295	229
Arity Standard Prolog	95	77
LPA microPROLOG *All Varieties*	CALL	CALL
MPROLOG Language Primer *LOGICWARE*	50	45
MPROLOG P500 *by LOGICWARE*	495	395
MPROLOG P550 *by LOGICWARE*	220	175
Prolog-86 *from Solution Systems*	125	88
Prolog-86 Plus *from Solution Systems*	250	199
Turbo PROLOG *by Borland Intl*	100	63
Turbo PROLOG Toolbox *by Borland Intl*	100	64

ai - smalltalk language

	List	Ours
Smalltalk/V *by Digitalk*	99	84
EGA/CGA Color Option	49	45
Goodies Diskette	49	45
Smalltalk/Comm	49	42

ai - texas instruments

	List	Ours
PC Scheme *Lisp*	95	84
Personal Consultant Easy	495	435
Personal Consultant Plus	2950	2589
Personal Consultant Runtime	95	84

apl language

	List	Ours
APL*PLUS/PC *by STSC*	595	424
APL*PLUS/PC Spreadsheet Mgr *by STSC*	195	139
APL*PLUS/PC Tools Vol 1 *by STSC*	295	199
APL*PLUS/PC Tools Vol 2 *by STSC*	85	58
ATLAS*GRAPHICS *by STSC* *New*	450	329
Financial/Statistical Library *by STSC*	275	189
Pocket APL *by STSC*	95	69
STATGRAPHICS *by STSC*	795	579

assembly language

	List	Ours
386 ASM/LINK *Cross Asm by Phar Lap*	495	389
8088 Assembler *w/Z-80 Translator by 2500 AD*	100	89
ASMLIB *Function Library by BC Assoc*	149	125
asmTREE *B-Tree Dev System by BC Assoc*	395	339
Cross Assemblers *Various by 2500 AD*	CALL	CALL
Microsoft Macro Assembler	150	93
Norton Utilities *by Peter Norton*	100	CALL
Norton Utilities (Advanced) *New*	150	99
screenplay *by Flexus*	100	79
Turbo EDITASM *by Speedware*	99	84
Uniware Cross Assemblers *Various by SDS*	CALL	CALL
Visible Computer: 8088 *by Software Masters*	80	64

basic language

	List	Ours
87 Software Pak *by Hauppauge*	180	149
EXIM Services Toolkit *by EXIM*	50	45
Finally *by Komputerwerks*	99	85
Inside Track *from Micro Help*	65	51
MACH 2 *by Micro Help*	69	55
MicroHelp Utilities *by MicroHelp* *New*	59	49
Microsoft QuickBASIC Compiler *New version*	99	63
Peeks 'n Pokes *from MicroHelp*	45	37
Professional BASIC *by Morgan*	99	68
8087 Math Support	50	42
Quick-Tools *by BC Associates* *New*	130	109
QuickPak *by Crescent Software*	69	59
Scientific Subroutine Library *by Peerless*	125	99
Screen Sculptor *by Software Bottling*	125	91
Stay-Res *by MicroHelp*	95	73
True Basic *w/Run-time by True Basic*	245	179
True Basic	150	97
Run-time Module	150	97
Various Utilities	50	41
Turbo BASIC Compiler *by Borland Intl*	100	64

blaise products

	List	Ours
ASYNCH MANAGER *Specify C or Pascal*	175	135
C TOOLS PLUS	175	135
EXEC *Program Chainer*	95	75
LIGHT TOOLS *for Datalight C*	100	79
PASCAL TOOLS	125	99
PASCAL TOOLS 2	100	79
PASCAL TOOLS & TOOLS 2	175	135
RUNOFF *Text Formatter*	50	45
TURBO ASYNCH PLUS	100	79
TURBO C TOOLS *New*	129	CALL
TURBO POWER TOOLS PLUS	100	79
VIEW MANAGER *Specify C or Pascal*	275	199

borland products

	List	Ours
EUREKA *Equation Solver*	100	64
Reflex & Reflex Workshop	200	128
Reflex *Data Base System*	150	89
Reflex Workshop	70	45
Sidekick & Traveling Sidekick	125	85
Sidekick	85	57
Traveling Sidekick	70	45
Superkey	100	64
Turbo BASIC Compiler	100	64
Turbo C Compiler	100	64
Turbo Database Toolbox	70	41
Turbo Editor Toolbox	70	41
Turbo Gameworks Toolbox	70	41
Turbo Graphix Toolbox	70	41
Turbo Jumbo Pack	300	219
Turbo Lightning	100	64
Turbo PASCAL Numerical Methods Toolbox	100	64
Turbo PASCAL and Tutor	125	85
Turbo PASCAL	100	64
Turbo Tutor	40	24
Turbo PROLOG Compiler	100	63
Turbo PROLOG Toolbox	100	64
Word Wizard	70	47
Word Wizard and Turbo Lightning	150	94

c++

	List	Ours
C++ *by Guidelines w/version 1.1 kernel*	195	172
PforCe++ *Function Library by Phoenix*	395	225

c compilers

	List	Ours
C86PLUS *by Computer Innovations*	497	359
Datalight C Compiler *by Datalight*	60	43
Datalight Developer Kit	99	74
Datalight Optimum-C	139	99
DeSmet C *w/Debugger & Large case* *New Version*	209	184
DeSmet C *w/Debugger only* *New Version*	159	138
Eco-C *with Graphics by Ecosoft*	125	83
Lattice C Compiler *vers. 3.2 from Lattice*	500	265
Mark Williams Let's C Combo Pack *New Version*	125	99
Let's C *Compiler* *New Version*	75	54
csd *Source Level Debugger* *New Version*	75	54
Microsoft C *with CodeView*	450	269
Turbo C Compiler *by Borland*	100	64
Uniware 68000/10/20 Cross Compiler *by SDS*	CALL	CALL

c interpreters

	List	Ours
C-terp *by Gimpel, Specify compiler*	300	219
C Trainer *with Book by Catalytix*	122	87
Instant C *by Rational Systems*	500	369
Introducing C *by Computer Innovations*	125	99
Run/C *by Age of Reason*	120	79
Run/C Professional *by Age of Reason*	250	157

c utilities

	List	Ours
c-tree & r-tree Combo *by FairCom*	650	529
c-tree *ISAM File Manager*	395	329
r-tree *Report Generator*	295	249
C Windows *by Syscom*	100	85
C Wings *by Syscom*	50	43
dBx *dBASE to C Translator by Desktop AI*	350	299
EASY SCREEN *by Retail Mgmt Systems* *New*	225	199
Flash-up Windows *by Software Bottling*	90	78
GraphiC *Color version by Sci Endeavors*	350	282
GRAFLIB *by Sutrasoft*	175	159
HALO *Graphics by Media Cybernetics*	300	205
HALO Development Pkg *for Microsoft*	595	389
The HAMMER *by OES Systems*	195	129
PANEL *Forms Management by Roundhill*	295	CALL
PANEL Plus *by Roundhill*	495	CALL
PC Lint *by Gimpel Software*	139	99
PLOTHI *by Sutrasoft*	175	159
PLOTHP *by Sutrasoft*	175	159
Professional C Windows *by Washburn*	89	79
Scientific Subroutine Library *by Peerless*	175	128
screenplay *for C by Flexus*	150	129
Vitamin C *by Creative Programming*	225	158
VC Screen *Forms Designer*	100	79
Zview *by Data Mgmt Consultants*	245	139

cobol language

	List	Ours
COBOLspII *by Flexus*	395	329
FPLIB *for Realia COBOL by BC Associates*	149	129
Micro Focus COBOL *See Micro Focus Section*		
Microsoft COBOL *See Microsoft Section*		
Realia COBOL with RealMENU *New*	1145	899
Realia COBOL	995	783
RealCICS	995	783
RM/COBOL *by Ryan-McFarland*	950	CALL
RM/COBOL 85 *by Ryan-McFarland*	1250	CALL
screenplay *for COBOL by Flexus*	175	139

css products

	List	Ours
PC/SPELL *by Custom Software Systems* *New*	49	45
PC/TOOLS *New*	49	45
PC/TOP *New*	49	45
PC/VI	149	99

debuggers & profilers

	List	Ours
386 DEBUG *Cross Debugger by Phar Lap*	195	129
Advanced Trace-86 *by Morgan Computing*	175	115
Codesifter *Profiler by David Smith*	119	85
Codesmith-86 *by Visual Age*	145	98
DSD87 *by Soft Advances*	125	79
MiniProbe *by Atron*	395	369
Periscope I *with Board by Periscope*	345	289
Periscope II *with NMI Breakout Switch*	175	139
Periscope II-X *Software only*	145	105
Periscope III *8 MHz version*	995	829
Periscope III *10 MHz version*	1095	899
The PROFILER *with Source Code by DWB*	125	89
TURBOsmith *Source debugger for Turbo Pascal*	69	59
The WATCHER *Profiler by Stony Brook*	60	51

disk utilities

	List	Ours
Back-It *by Gazelle Systems* *New*	100	89
Disk Optimizer *by Softlogic Systems*	60	55
FASTBACK *by 5th Generation Systems*	179	129
XenoCopy-PC *by XenoSoft* *New*	80	69

dos utilities

	List	Ours
Command Plus *by ESP Software*	80	69
FANSI-CONSOLE *by Hersey Micro*	75	62
Norton Commander *by Peter Norton*	75	CALL
OPAL *Shell Language by Software Factory* *New*	99	89
OPART *Screen Painter* *New*	75	67
Q-DOS II *by Gazelle Systems* *New*	70	59
Scroll & Recall *by Opt-Tech Data*	69	59
Taskview *by Sunny Hill Software*	80	55

essential products

Turbo C versions available.

	List	Ours
C Essentials *by Essential Software*	100	75
C Utility Library	185	119
Essential Comm Library *with Debugger*	250	189
Essential Comm Library *Software Only*	185	125
Breakout Debugger Only *Any language*	125	89
Essential Graphics	250	183

forth language

	List	Ours
CFORTH *Native Code Compiler by LMI*	300	229
Forth/83 Metacompiler *Specify Target*	750	599
PC/Forth *by Laboratory Microsystems*	150	109
PC/Forth+ *by Laboratory Microsystems*	250	199
Advanced Color Graphics Support	100	74
Enhanced Graphics Support	200	148
Intel 8087 Support	100	74
Interactive Symbolic Debugger	100	74
Native Code Optimizer	200	148
Software Floating Point	100	74
UR/Forth *by LMI*	350	279
Object Module Libraries	500	395

fortran language

	List	Ours
50 MORE: FORTRAN *by Peerless Engr*	125	95
ACS Time Series *by Alpha Computer Service*	495	389
Essential Graphics *by Essential Software*	250	183
For-Winds *by Alpha Computer Service*	90	69
Forlib-Plus *by Alpha Computer Service*	70	44
FORTLIB *by Sutrasoft*	125	109
FORTRAN Addendum *by Impulse Engr*	95	85
FORTRAN Addenda *by Impulse Engr*	165	138
GRAFLIB *by Sutrasoft*	175	159
HALO *Graphics by Media Cybernetics*	300	205
I/O PRO *by MEF Environmental*	149	129
Microcompatibles Combo Package	240	215
Grafmatic	135	117
Plotmatic	135	117
Microsoft FORTRAN *w/CodeView*	450	269
No Limit *by MEF Environmental*	129	109
Numerical Analyst *by MAGUS*	295	249
PANEL *by Roundhill Computer Systems*	295	CALL
PLOTHI *by Sutrasoft*	175	159
PLOTHP *by Sutrasoft*	175	159
RM/FORTRAN *by Ryan-McFarland*	595	CALL
RTC PLUS *Fortran to C by Cobalt Blue*	325	289
Scientific Subroutine Lib *by PSI/Systems*	175	128
Statistician *by Alpha Computer Service*	295	235
Statlib.GL: *by PSI/Systems*	295	239
Statlib.TSF: *by PSI/Systems*	295	239
Strings & Things *by Alpha Computer Service*	70	45

greenleaf products

	List	Ours
Greenleaf Comm Library	185	125
Greenleaf Data Windows	225	157
with Source Code	450	289
Greenleaf Functions	185	125

help utilities

	List	Ours
HELP/Control *by MDS*	125	99
On-line Help *from Opt-Tech*	149	99
SoftScreen/HELP *by Dialectic Systems*	195	149

lattice products

	List	Ours
Lattice C Compiler *ver 3.2 from Lattice*	500	265
with Library Source Code	900	495
C Cross Reference Generator	50	37
with Source Code	200	139
C-Food Smorgasbord *Function Library*	150	95
with Source Code	300	179
C-Sprite *Source Level Debugger*	175	119
Curses *Screen Manager*	125	85
with Source Code	250	169
dBC *Specify dBC II or dBC III*	250	169
with Source Code	500	356
dBC III Plus	750	594
with Source Code	1500	1184
LMK *Make Facility*	195	138
RPG II Combo *All three items below*	1100	875
RPG II Compiler *No Royalties*	750	625
RPG II SEU *Screen Entry Utility*	250	199
RPG II Sort/Merge	250	199
RPG II Screen Design Aid Utility	350	309
SecretDisk *File Encryption Utility*	120	88
SideTalk *Resident Communications*	120	88
SSP/PC *Scientific Subroutine Library*	350	269
Text Management Utilities	120	88
TopView Toolbasket *Function Library*	250	178
with Source Code	500	356

metagraphics products

	List	Ours
LightWINDOW/C *for Datalight C*	95	79
MetaFONTS	95	79
MetaFONTS/PLUS	275	229
MetaWINDOW *No Royalties*	195	159
MetaWINDOW/PLUS	275	229
TurboWINDOW/C *for Turbo C*	95	79
TurboWINDOW/Pascal *for Turbo Pascal*	95	79

micro focus products

	List	Ours
Micro Focus Level II COBOL *w/Animator* *New*	495	395
Level II COBOL	349	279
Level II Animator	195	155
Micro Focus Level II COBOL/ET for UNIX *New*	CALL	CALL
Micro Focus Personal COBOL	149	119
Micro Focus Professional COBOL	2000	1595
Micro Focus VS COBOL/XENIX *New*	1495	1195

CIRCLE 129 ON READER SERVICE CARD

Micro Focus Support Products:		
COBOL/IQ *Ad hoc Report Writer* New	495	395
COBOL/IQ *for DOS 3.X Networks* New	995	795
FORMS-2	295	235
SOURCEWRITER	995	795

microport products

System V/AT *by Microport Systems*	549	475
Runtime System *(Operating System)*	199	189
Software Development System	249	235
Text Preparation System	199	189
Unlimited License Kit	249	235

microsoft products

Microsoft BASIC Compiler *for XENIX* New	695	419
Microsoft BASIC Interpreter *for XENIX*	350	209
Microsoft C Compiler *with CodeView*	450	269
Microsoft COBOL Compiler *with COBOL Tools*	700	429
for XENIX	995	609
Microsoft FORTRAN *Optimizing Compiler/CodeView*	450	269
Microsoft FORTRAN *for XENIX*	695	419
Microsoft Learning DOS	50	36
Microsoft LISP *Common LISP*	250	149
Microsoft MACH 10 *with Mouse & Windows*	549	369
Microsoft MACH 10 *Board only*	399	279
Microsoft Macro Assembler	150	93
Microsoft Mouse *Bus Version*	175	114
Microsoft Mouse *Serial Version*	195	124
Microsoft muMath *Includes muSIMP*	300	179
Microsoft Pascal Compiler	300	179
for XENIX	695	419
Microsoft QuickBASIC Compiler *New version*	99	63
Microsoft Sort	195	125
Microsoft Windows	99	63
Microsoft Windows Development Kit	500	299
Microsoft Word *Special Price thru July*	450	225

modula-2 language

MODULA-2 Apprentice Pkg by LOGITECH	99	79
MODULA-2 Magic Pkg by LOGITECH	99	79
MODULA-2 ROM Pkg & Cross RT Debugger	299	239
MODULA-2 Window Pkg by LOGITECH	49	39
MODULA-2 Wizard's Pkg by LOGITECH	199	159
REPERTOIRE *for MODULA-2 by PMI*	89	79
Object Code Only	19	15

mouse products

LOGIMOUSE BUS *with PLUS Pkg by LOGITECH*	119	98
with PLUS & PC Paintbrush	149	119
with PLUS & CAD Software	189	153
with PLUS & CAD & Paint	219	179
LOGIMOUSE C7 *with PLUS Pkg, Specify Connector*	119	98
with PLUS & PC Paintbrush	149	119
with PLUS & CAD Software	189	153
with PLUS & CAD & Paint	219	179
Microsoft Mouse *Bus Version*	175	114
Microsoft Mouse *Serial Version*	195	124

other languages

CCS MUMPS *Single-User by MGlobal*	60	50
CCS MUMPS *Single-User Multi-Tasking*	150	129
CCS MUMPS *Multi-User*	450	359
Janus/ADA C Pak *by R&R Software*	95	84
Janus/ADA D Pak *by R&R Software*	900	769
Janus/ADA ED Pak *by R&R Software*	395	349
Marshal Pascal *by Marshal Language Systems*	189	155
Pascal-2 *by Oregon Software* New	395	325
Personal REXX *by Mansfield Software*	125	99
SNOBOL4+ *by Catspaw*	95	80

other products

Dan Bricklin's Demo Pgm *by Software Garden*	75	57
Dan Bricklin's Demo Tutorial	50	45
Informix *All Varieties by Informix*	CALL	CALL
Instant Replay *by Nostradamus* New Version	150	CALL
MKS Toolkit *with vi Editor by MKS*	139	114
MicroTEX *Typesetting from Addison Wesley* New	295	CALL
Net-Tools *by BC Associates*	149	129
OPT-Tech Sort *by Opt-Tech Data Proc*	149	99
PC/TOOLS *by Custom Software* New	49	45
Screen Machine *by MicroHelp*	79	59
VTEK *Term Emulator by Sci Endeavors*	150	129

phoenix products

Pasm86 *Macro Assembler version 2.0*	195	108
Pdisk *Hard Disk & Backup Utility*	145	99
Pfantasy Pac *Phoenix Combo*	995	619
Pfinish *Execution Profiler*	395	225
Pfix86plus *Symbolic Debugger*	395	225
PforCe *Specify C Compiler*	395	225
PforCe++ *Specify C Compiler and C++*	395	225
Plink86plus *Overlay Linker*	495	299
Pmaker *Make Utility*	125	78
Pmate *Macro Text Editor*	195	108
Pre-C *Lint Utility*	295	154
Ptel *Binary File Transfer Program*	195	108

polytron products

PolyBoost *The Software Accelerator*	80	64
PolyDesk III New	99	72
PolyDesk III Archivist New	50	42
PolyDesk III Cryptographer New	50	42
PolyDesk III Talk New	70	52
PolyLibrarian *Library Manager*	99	73
PolyLibrarian II *Library Manager*	149	109
PolyMake *UNIX-like Make Facility*	149	109
PolyShell	149	109
Polytron C Beautifier	50	42
Polytron C Library I	99	72
Polytron PowerCom *Communications*	139	105
PolyXREF *Complete Cross Ref Utility*	219	169
PolyXREF *One language only*	129	99
PVCS Network *Version Control System* New	1000	CALL
PVCS Corporate	395	309
PVCS Personal	149	109
PVMFM *Polytron Virtual Memory File Mgr* New	199	155

program mgmt utilities

Compact Source Print *by Aldebaran*	55	44
Interactive EASYFLOW *by Haventree*	150	125
PrintQ *by Software Directions*	89	84
Quilt Computing Combo Package New Version	250	199
QMake *Program Rebuild Utility*	99	79
SRMS *Software Revision Mgmt Sys* New Version	185	159
Source Print *by Aldebaran Labs*	75	59
TLIB *by Burton Systems Software*	100	89
Tree Diagrammer *by Aldebaran Labs*	55	49

raima products

dbQUERY *Single-User Query Utility*	195	129
Single-User with Source Code	495	389
Multi-User	495	389
Multi-User with Source Code	990	699
dbVISTA *Single-User DBMS*	195	129
Single-User with Source Code	495	389
Multi-User	495	389
Multi-User with Source Code	990	699

sco products

Complete XENIX System V *by SCO*	1295	994
Development System	595	499
Operating System *Specify XT or AT*	595	499
Text Processing Package	195	144
Lyrix *by SCO*	595	449
XENIX-NET *by SCO*	595	495
SCO Professional *1-2-3 Workalike for XENIX*	795	595

softcraft products

Btrieve *ISAM Mgr with No Royalties*	245	184
Xtrieve *Query Utility*	245	184
Report Option *for Xtrieve*	145	99
Btrieve/N *for Networks*	595	454
Xtrieve/N	595	454
Report Option/N *for Xtrieve/N*	345	269

text editors

Brief & dBrief Combo *from Solution Systems*	250	CALL
Brief *New version*	195	CALL
dBrief *Customizes Brief for dBASE III*	95	CALL
Epsilon *Emacs-like editor by Lugaru*	195	147
KEDIT *by Mansfield Software*	125	98
Micro/SPF *by Phaser Systems*	175	139
Microsoft Word *Special Price thru July*	450	225
PC/VI *by Custom Software Systems*	149	99
SPF/PC *by Command Technology Corp*	CALL	CALL
Vedit *by CompuView*	150	98
Vedit Plus *by CompuView*	185	128

turbo pascal utilities

ALICE *Interpreter by Software Channels*	95	66
DOS/BIOS & Mouse Tools *by Quinn-Curtis*	75	67
Flash-up Windows *by Software Bottling*	90	78
MACH 2 *for Turbo Pascal by Micro Help*	69	55
MetraByte D/A Tools *by Quinn-Curtis*	100	89
Science & Engrg Tools *by Quinn-Curtis*	75	67
Screen Sculptor *by Software Bottling*	125	91
screenplay *for Turbo Pascal by Flexus*	100	79
Speed Screen *by Software Bottling*	35	32
System Builder *by Royal American*	130	115
IMPEX *Query Utility*	75	69
Report Builder	75	69
TDebugPLUS *by TurboPower Software*	60	49
Turbo EXTENDER *by TurboPower Software*	85	64
Turbo OPTIMIZER *by TurboPower* New	75	65
Turbo OPTIMIZER *with Source Code* New	125	108
Turbo Professional *by Sunny Hill*	70	45
TurboHALO *from IMSI*	129	98
TurboPower Utilities *by TurboPower*	95	78
TurboRef *by Gracon Services*	50	45
TURBOsmith *Source Debugger by Visual Age*	69	59
Universal Graphics Library *by Quinn-Curtis* New	150	119

wendin products

Operating System Toolbox *Rebate Offer*	99	75
PCNX *Operating system* *Rebate Offer*	99	75
PCVMS *Similar to VAX/VMS*	99	75
XTC *Text Editor w/Pascal source*	99	75

xenix/unix products

Btrieve *ISAM File Mgr by SoftCraft*	595	454
C-terp *by Gimpel*	498	379
c-tree *ISAM Mgr by FairCom*	395	329
dBx *with Library Source by Desktop AI*	550	489
DOSIX *Console Version by Data Basics*	399	349
DOSIX *User Version by Data Basics*	199	179
Informix *All Varieties by Informix*	CALL	CALL
Micro Focus Products *See Micro Focus Section*		
Microport Products *See Microport Section*		
Microsoft Products *See Microsoft Section*		
PANEL Plus *by Roundhill Computer Systems*	495	CALL
REAL-TOOLS *Binary Version by PCT*	149	89
Library Source Version	399	289
Complete Source Version	499	369
RM/COBOL *by Ryan-McFarland*	1250	CALL
RM/FORTRAN *by Ryan-McFarland*	750	CALL
SCO Products *See SCO Section*		

LOWEST PRICES

Due to printing lead times, some of our current prices may differ from those shown here. Call for latest pricing.

FREE SHIPPING

Orders within the USA (including Alaska & Hawaii) are shipped FREE via UPS. Express shipping is available at the shipping carrier's standard rate with no rush fees or handling charges. To avoid delays when ordering by mail, please call first to determine the exact cost of express shipping.

CREDIT CARDS

VISA, MasterCard and Discover Card are accepted at no extra cost. Your card is charged when your order is shipped. Mail orders please include credit card expiration date and authorized signature.

CODs AND POs

CODs and Purchase Orders are accepted at no extra cost. POs with net 30-day terms are available to qualified US accounts only.

SALES TAX

Orders outside of Ohio are not charged state sales tax. Ohio customers please add 6% Ohio tax or provide proof of tax-exemption.

INTERNATIONAL ORDERS

Shipping charges for International and Canadian orders are based on the shipping carrier's standard rate. Since rates vary between carriers, please call or write for the exact cost. International orders (except Canada), please include an additional $10 for export preparation. All payments must be made with US funds drawn on a US bank. Please include your telephone number when ordering by mail. Due to government regulations, we cannot ship to all countries.

VOLUME ORDERS

Volume orders may qualify for additional discounts. Call us for special pricing.

SOUND ADVICE

Our knowledgeable technical staff can answer technical questions, assist in comparing products and send you detailed product information tailored to your needs.

30-DAY GUARANTEE

Most of our products (excluding books) come with a 30-day documentation evaluation period or a 30-day return guarantee. Please note that some manufacturers restrict us from offering guarantees on their products. Call for more information.

MAIL ORDERS

Please include your telephone number on all mail orders. Be sure to specify computer, operating system and any applicable compiler or hardware interface(s). Send mail orders to:

Programmer's Connection
Order Processing Department
136 Sunnyside Street
Hartville, OH 44632

CALL TOLL FREE

USA 800-336-1166
CANADA 800-225-1166
OHIO & ALASKA (Collect) .. 216-877-3781

TELEX 9102406879
EASYLINK 62806530

INTERNATIONAL 216-877-3781
CUSTOMER SERVICE .. 216-877-1110

Hours: Weekdays 8:30 AM to 8:00 PM EST.

Call or write for our FREE comprehensive price guide.

programmer's connection

Listing Three *(Text begins on page 42.)*

```
{***********************************************}
{***            Listing Three                ***}
{***      Multitasking Demonstration         ***}
{***           support routines              ***}
{***                                         ***}
{***              written by                 ***}
{***           Craig A. Lindley              ***}
{***                                         ***}
{***    Ver: 1.0     Last update: 03/11/87   ***}
{***                                         ***}
{***********************************************}

CONST

   COM1        = 1;        {com one PC port}
   portaddress = $3f8;     {address of UART for}
                           {COM1}

TYPE

   parity_type = (odd,even,none);

VAR

   regs:   register_type;


PROCEDURE Int14(portnumber,command,
                parameter:integer);

{Procedure to initialize the com ports}

BEGIN

   WITH regs DO
   BEGIN
      dx := portnumber - 1;
      ah := command;
      al := parameter;
      flags := 0;
      intr($14,regs);
   END;

END;


PROCEDURE Setserial(baudrate,stopbits,
                    databits: integer;
                    parity: parity_type);

{Configure COM1 with the specified parameters}

VAR

    parameter: integer;

BEGIN

   writeln('Configuring the serial parameters');
   writeln;
   CASE baudrate OF
      300: baudrate := 2;
     1200: baudrate := 4;
      ELSE baudrate := 4;   {default is 1200 baud}
   END;

   IF stopbits = 2 THEN
      stopbits := 1
   ELSE
      stopbits := 0;        {default is 1 stop bit}

   IF databits = 7 THEN
      databits := 2
   ELSE
      databits := 3;        {default is 8 bit words}

   parameter := (baudrate SHL 5)+
                (stopbits SHL 2)+databits;
```


(continued on page 73)

UNIFY DBMS/DOS. The UNIX World Leader Brings A New Dimension To DOS Application Development.

What happens as the DOS world expands? As a new generation of hardware takes over? As networking becomes more important? The potential is enormous. But until now, the tools to achieve it have been limited.

Now a leader from another world unleashes that potential: UNIFY® DBMS. The leading relational DBMS in the UNIX™ world. And now, the most advanced set of application development tools in the DOS world.

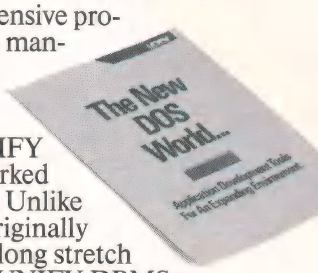
With UNIFY DBMS, DOS developers have new power to build more sophisticated applications than ever before possible.

The power to write high performance "C" programs that will access the data base, using Unify's Direct Host Language Interface.

The power of an industry standard query language—SQL.

The power of unmatched speed in production applications. Only UNIFY DBMS is specifically engineered for transaction throughput. With unique performance features like PathFinder™ Architecture multiple access methods, for the fastest possible data base access.

The power of comprehensive program development and screen management tools. Plus a state-of-the-art fourth generation report-writer.



What's more, with UNIFY DBMS, the potential of networked applications becomes a reality. Unlike DBMS systems which were originally single-user (and which have a long stretch to accommodate more users), UNIFY DBMS is a *proven* multi-user system.

And because UNIFY DBMS/DOS is the best of two worlds, it offers you the most powerful benefit of all: DBMS applications that can grow as your needs grow. From single user DOS. To networked DOS. To multi-user UNIX. All without changing your applications.

Call the Unify Information Hotline for our free booklet: The New DOS World. (503) 635-7777

UNIFY
CORPORATION
4000 Kruse Way Place
Lake Oswego, OR 97034

CIRCLE 332 ON READER SERVICE CARD

On April 2, 1987 IBM and Quarterdeck announced the next generation in personal computing.

Introducing DESQview 2.0. Improving the past and ready for the future right now.

In one sweeping announcement from Miami Beach and New York City, IBM established new standards of performance for personal computers, with its new Personal System/2.™ Quarterdeck was there with IBM and simultaneously helped establish new standards for multi-tasking and multi-windowing.

We were there for them then. We're here for you now.

If you use two or more software programs, if you use a PC-compatible machine, if you own a new 80386 computer, if you've just bought one of the new Personal System/2 computers, or if you've tried Microsoft Windows and were disappointed but still need the power of graphics programs, DESQview 2.0 is the answer.

Consider this. InfoWorld voted DESQview's earlier version 1986 Product of the Year. SoftSector gave it the Editor's Choice Award. In PC Tech Journal's "System Builder Contest" at Comdex Fall it was voted best operating environment. And 450,000 dedicated users on four continents have voted yes with their dollars.

The new DESQview 2.0 is an order of magnitude better.

This unique software program enhances the power of your personal computer and makes it more convenient to use. It still gives your PC the power of many PCs. It still does windows. It still multi-tasks. It still breaks the DOS 640K barrier. It still transfers data. It still dials your phone. It still gives you menus for DOS. It still remembers your keystrokes (macros). It still runs your existing programs and your new programs soon to come. In fact now you can even run Windows-, GEM-, and Topview-specific programs too. And with 386 machines and our Expanded Memory Manager it still becomes a 386 control program, but now you can run text *and* CGA graphics programs in background.

The new DESQview 2.0.

For us it's the next logical step.

For you it's windows of opportunity.

SYSTEM REQUIREMENTS
▪ IBM Personal Computer and 100% compatibles (with 8086, 8088, 80286 or 80386 processors) with monochrome or color display; IBM Personal System/2 ▪ Memory: 640K recommended; for DESQview itself 0-145K ▪ Expanded Memory (Optional): expanded memory boards compatible with the Intel AboveBoard; enhanced expanded memory boards compatible with the AST RAMpage ▪ Disk: Two diskette drives or one diskette drive and a hard disk ▪ Graphics Card (Optional): Hercules, IBM Color/Graphics (CGA), IBM Enhanced Graphics (EGA), IBM Personal System/2 Advanced Graphics (VGA) ▪ Mouse (Optional): Mouse Systems, Microsoft and compatibles ▪ Modem for Auto-Dialer (Optional): Hayes or Compatible ▪ Operating System: PC-DOS 2.0–3.3; MS-DOS 2.0–3.2 ▪ Software: Most PC-DOS and MS-DOS application programs; programs specific to TopView 1.1, GEM 1.1 and Microsoft Windows 1.03 ▪ Media: DESQview 2.0 is available on either 5¼″ or 3½″ floppy diskettes

Rush me DESQview 2.0! Today!

No. of Copies	Media 3½″/5¼″	Product		Retail Price ea.	Total
		DESQview 2.0		$129.95	$
	Shipping & Handling		USA Outside USA	$ 5.00 $ 10.00	$
		Sales Tax (CA residents)		6.5%	$
				Amount Enclosed	$

Payment: ☐ Visa ☐ MC ☐ AMEX ☐ Check

Credit Card: Valid Since ______ / ______ Expiration ______ / ______

Card Number: ________________

Credit Card Name ________________

Shipping Address ________________

City ________ State ________ Zip ________ Telephone ________

Mail to: Quarterdeck Office Systems, 150 Pico Boulevard, Santa Monica, CA 90405

NOTE: If you own DESQview call us for a special upgrade offer, or send in your DESQview registration card and we'll send you upgrade information.

DESQ™ view
Quarterdeck

Quarterdeck Office Systems • 150 Pico Boulevard, Santa Monica, CA 90405 • (213) 392-9851

CIRCLE 284 ON READER SERVICE CARD

Listing Three *(Listing continued, text begins on page 42.)*

```
   CASE parity OF
      odd: parameter := parameter + 8;
     even: parameter := parameter + 24;
     none: ;
   END;

   Int14(COM1,0,parameter);{do the configuration}

END;


FUNCTION Serialstatus : integer;

{Get the status of COM1 port}

BEGIN

   Int14(COM1,3,0);
   serialstatus := regs.ax;

END;


PROCEDURE Serialout(b:byte);

{Send a byte out the COM1 port}

BEGIN

   {wait till UART is ready}
   WHILE (Serialstatus AND $2000) = 0 DO;

   {then send the byte out}
   port[portaddress] := b;

END;



PROCEDURE Enable_serial_int;

BEGIN

{clear the serial interface of any garbage}

 INLINE($BA/portaddress  /$EC/
        $BA/portaddress+5/$EC/
        $BA/portaddress+6/$EC);

 INLINE($E4/$21/$24/$EF/$E6/$21); {IRQ4 enabled}
 port[portaddress+4] := $0B;      {set DTR, RTS}
                                  {and OUT2}
 port[portaddress+1] := 1;        {receiver}
                                  {interrupt}
                                  {enabled}
END;

{End of RS-232 procedures}
```

End Listings

Listing One *(Text begins on page 94.)*

```
                    Listing 1 -- vbios.c

#include <stdio.h>
#include <dos.h>   /* (Microsoft file) includes for int86() */

/*   VBIOS.C: Various cursor and i/o routine using
 *   the bios interrupts (see below for greater detail):
 *-------------------------------------------------------
 *   Copyright (C) 1987 Allen I. Holub. All rights reserved.
 *-------------------------------------------------------
 * Externally accessable routines:
 *
 * int   vb_getpage ()            Get active video page #
 * void  vb_putchar (c)           write a single character
 * void  vb_getchar (c)           get a key from the bios.
 * void  vb_puts    (s, move)     write a string
 * void  vb_replace (c)           write char w/o moving cursor
 * int   vb_inchar  (attrib)      Get character & attribute
 *
 * void  vb_setcur  (posn)        Set cur pos as int on cur page
 * int   vb_getcur  ()            Get cur pos as int from cur page
 * void  vb_ctoyx (y,x)           Set cursor position to (y,x)
 * void  vb_getyx (&y, &x)        Get cursor position
 *
 * int   vb_iscolor()             color monitor installed
 * void  vb_cursize (top,bot)     Set cursor size
 * void  vb_blockcur()            make a block cursor
 * void  vb_normalcur()           revert to a normal cursor
 *
 * void  vb_scroll(l,r,t,b,a)     Scroll region
 */

/*------------------------------------------------------*/

extern int      int86( int, union REGS *, union REGS *);

/*------------------------------------------------------*/

#define VIDEO_INT     0x10    /* Video interrupt          */
#define KB_INT        0x16    /* Keyboard interrupt       */

#define CUR_SIZE      0x1     /* Set cursor size          */
#define SET_POSN      0x2     /* Modify cursor posn       */
#define READ_POSN     0x3     /* Read current cursor posn */
#define WRITE         0x9     /* Write character          */
#define WRITE_TTY     0xe     /* Write char & move cursor */
#define GET_VMODE     0xf     /* Get video mode & disp pg */

/*------------------------------------------------------*/

static union REGS       Regs;       /* Used to talk to DOS */
static int              Attribute;  /* Current attribute   */

/*------------------------------------------------------*/

void    vb_scroll( x_left, x_right, y_top, y_bottom, amt )
{
        /* Scroll the indicated region on the screen.
         * If amt is negative, scroll down.
         */

        if( amt < 0 )
        {
            Regs.h.ah = 7    ;
            Regs.h.al = -amt ;
        }
        else
        {
            Regs.h.ah = 6   ;
            Regs.h.al = amt ;
        }

        Regs.h.bh = 0x07    ;
        Regs.h.cl = x_left  ;
        Regs.h.ch = y_top   ;
        Regs.h.dl = x_right ;
        Regs.h.dh = y_bottom ;
        int86(0x10, &Regs, &Regs);
}

/*------------------------------------------------------*/

int     vb_inchar( attrib )
int     *attrib;
{
        /* Return the character at the current cursor
         * position and, if attrib is non-NULL, put the
         * attribute there. Note that vb_getpage() will mess
         * up the fields in the Regs structure so it must
         * be called first.
         */

        Regs.h.bh = vb_getpage() ;
        Regs.h.ah = 8 ;

        int86( VIDEO_INT, &Regs, &Regs );

        if( attrib )
                *attrib = Regs.h.ah & 0xff ;

        return( Regs.h.al & 0xff );
}

/*------------------------------------------------------*/

int     vb_getpage()
{
        /* Returns the currently active display page number
```

```
        */

        Regs.h.ah = GET_VMODE;
        int86( VIDEO_INT, &Regs, &Regs );

        return (int) Regs.h.bh ;
}

/*--------------------------------------------------------*/

void    vb_cursize( top_line, bot_line )
{
        /* Scan lines are numberd 0 at the top and 7 at the
         * bottom on the color card. On the monochrome card
         * they're 0-12. If top & bot are reversed you'll
         * get a 2 part cursor. Top_line determines the
         * position of the top scan line of the cursor,
         * bot_line is the bottom. A normal cursor can be
         * created with vb_cursize(6,7). Cursize(0,7) will
         * fill the entire area occupied by a character.
         * Cursize(0,1) will put a line over the character
         * rather than under it.
         */

        Regs.h.ch = top_line ;
        Regs.h.cl = bot_line ;
        Regs.h.ah = CUR_SIZE ;
        int86( VIDEO_INT, &Regs, &Regs );
}

/*--------------------------------------------------------*/

int vb_iscolor()   /* Returns true if a color card is active */
{
        Regs.h.ah = GET_VMODE ;
        int86( VIDEO_INT, &Regs, &Regs );
        return( Regs.h.al != 7 );
}

void vb_blockcur()      /* Make the cursor a block curser */
{
        vb_cursize( 0, vb_iscolor() ? 7 : 12 );
}

void vb_normalcur()      /* Make it an underline cursor */
{
        if( vb_iscolor() )
                vb_cursize(  6,  7 );
        else
                vb_cursize( 11, 12 );
}

/*--------------------------------------------------------*/

void    vb_setcur( posn )
int     posn;
{
        /* Modify current cursor position. The top byte of
         * "posn" value holds the row (y), the bottom byte,
         * the column (x). The top-left corner of the screen
         * is (0,0). Pagenum is the video page number. Note
         * that vb_getpage() will mess up the fields in the
         * Regs structure so it must be called first.
         */

        Regs.h.bh = vb_getpage() ;
        Regs.x.dx = posn          ;
        Regs.h.ah = SET_POSN      ;
        int86( VIDEO_INT, &Regs, &Regs );
}

int     vb_getcur()
{
        /* Get current cursor position. The top byte of the
         * return value holds the row, the bottom by the
         * column. Pagenum is the video page number. Note
         * that vb_getpage() will mess up the fields in the
         * Regs structure so it must be called first.
         */

        Regs.h.bh = vb_getpage() ;
        Regs.h.ah = READ_POSN    ;
        int86( VIDEO_INT, &Regs, &Regs );
        return( Regs.x.dx );
}

/*---------------------------------------------------------
 * vb_cotyx() and vb_getyx also get the cursor position.
 * They use x and y values, however.
 */

void    vb_ctoyx ( y, x )
{
        vb_setcur( (y << 8) | (x & 0xff) );
}

void    vb_getyx( yp, xp )
int     *yp, *xp;
{
        register int    posn;

        posn = vb_getcur();
        *xp  = posn         & 0xff ;
        *yp  = (posn >> 8)  & 0xff ;
}

/*--------------------------------------------------------*/
```

(continued on page 78)

THE PROGRAMMER'S SHOP

helps save time, money and cut frustrations. Compare, evaluate, and find products.

RECENT DISCOVERY

Periscope III - debugger with 64K protected RAM and breakout switch; breakpoints for hardware, memory, port, data. Real-time trace buffer, pass counter. PC $ 829

AI-Expert System Dev't

Arity Combination Package	PC	$ 979
System - use with C	MS	$ 229
SQL Dev't Package	MS	$ 229
Auto-Intelligence	PC	$ 739
Experteach - Powerful, samples	PC	$ 339
Exsys	PC	$ 309
Runtime System	PC	$ 469
Insight 2+	MS	$ 379
Intelligence/Compiler	PC	$ 739
T.I. - PC Easy	PC	$ 435
Personal Consultant Plus	PC	$2589
Personal Consultant Runtime	PC	$ 85
Turbo Expert-Startup-(400 rules)	PC	$ 129
Corporate (4000 rules)	PC	$ 359

AI-Lisp

Microsoft MuLisp 85	MS	$ 159
PC Scheme LISP - by TI	PC	$ 85
TransLISP - learn fast	MS	$ 89
TransLISP PLUS		
Optional Unlimited Runtime		$ 139
PLUS for MSDOS		$ 179

Others: IQ LISP ($155), IQC LISP ($269)

AI Prolog

APT - Active Prolog Tutor - build applications interactively	PC	$ 49
ARITY Prolog - Interpreter	PC	$ 229
COMPILER/Interpreter-EXE	PC	$ 569
Standard Prolog	MS	$ 77
MicroProlog - Prof. Entry Level	MS	$ 85
MicroProlog Prof. Comp./Interp.	MS	$ 439
MPROLOG P550	PC	$ 175
Prolog-86 - Learn Fast	MS	$ 89
Prolog-86 Plus - Develop	MS	$ 229
TURBO PROLOG by Borland	PC	$ 69

Basic

BAS_C - economy	MS	$ 179
BAS_PAS - economy	MS	$ 135
Basic Development System	PC	$ 105
Basic Development Tools	PC	$ 89
Basic Windows by Syscom	PC	$ 95
BetterBASIC	PC	$ 129
Exim Toolkit - full	PC	$ 39
Finally - by Komputerwerks	PC	$ 85
Inside Track	PC	$ 50
Mach 2 by MicroHelp	PC	$ 55
Peeks n Pokes	PC	$ 35
QuickBASIC	PC	$ **69**
Stay-Res	PC	$ 75
Turbo BASIC - by Borland	PC	$ 69

Note: All prices subject to change without notice. Mention this ad. Some prices are specials. Ask about COD and POs. Formats: 3" laptop now available, plus 200 others. UPS surface shipping add $3/item.

FREE Newsletter

Insightful commentary, guest columnists, survey results, and valuable resource listings. Interviews, technical articles, predictions — even cartoons. No wonder 96% of our readers pass *The Programmer's Letter* on to their friends; no wonder 72% make sure they get their copy back to keep for reference! You can request a FREE sample copy today by calling our toll-free number. A personal subscription is just $25 per year.

Our Services:

- Programmer's Referral List
- Compare Products
- Help find a Publisher
- Evaluation Literature FREE
- BBS - 7 PM to 7 AM 617-740-2611
- Dealers Inquire
- Newsletter
- Rush Order
- Over 700 products
- National Accounts Center

C Language-Compilers

AZTEC C86 - Commercial	PC	$499
C86 PLUS - by CI	MS	$379
Datalight C - fast compile, good code, 4 models, Lattice compatible, Lib source. Dev'rs Kit	PC	$ 77
Datalight Optimum - C	MS	$109
with Light Tools by Blaise	PC	$168
Lattice C - from Lattice	MS	$269
Let's C Combo Pack	PC	$ 99
Let's C	PC	$ 57
Microsoft C 4.0- Codeview	MS	$**275**
Rex - C/86 by Systems & Software - standalone ROM	MS	$695
Turbo C by Borland	PC	$ 69
Uniware 68000/10/20 Cross Compiler	MS	Call
Wizard C	MS	$299
Rom Development Package	MS	$259

C Language-Interpreters

C-terp by Gimpel - full K & R	MS	$219
C Trainer - by Catalytix	PC	$ 89
INSTANT C - Source debug, Edit to Run-3 seconds, .OBJs	MS	$369
Interactive C by IMPACC Assoc.	PC	$209
Run/C Professional	MS	$155
Run/C Lite	MS	$ 79

C Libraries-General

Blackstar C Function Library	PC	$ 79
C Essentials - 200 functions	PC	$ 75
C Function Library	MS	$109
C Tools Plus (1 & 2) - Blaise	PC	$125
C Utilities by Essential	PC	$129
C Worthy Library - Complete, machine independent	MS	$249
Entelekon C Function Library	PC	$119
Entelekon Superfonts for C	PC	$ 45
Greenleaf Functions-portable, ASM		$139
LIGHT TOOLS by Blaise	PC	$ 69

FEATURE

UI Programmer - Quickly generate dBASE User Interfaces, prototypes. Use supplied templates or create own. Pop-up help, bounce bar menus, screen forms. II, III, FoxBASE+, Quicksilver, Clipper. PC $249

RECENT DISCOVERY

CxPERT - Expert systems shell, translates to C code to integrate with your application. Certainty factors, explanations, inheritance, frames, help. MS $295

dBASE Language

Clipper compiler	PC	Call
dBASE II	MS	$329
dBase III Plus	PC	$429
dBASE III LanPack	PC	$649
DBXL Interpreter by Word Tech	PC	$139
FoxBASE+ - single user	MS	$349
Quick Silver by Word Tech	PC	$499

dBASE Support

dBase Tools for C	PC	$ 65
dBrief with Brief	PC	Call
DBC ISAM by Lattice	MS	Call
dBx Translator to C	MS	$319
dFlow - flowchart, xref	MS	Call
Documentor - dFlow superset	MS	Call
Genifer by Bytel-code generator	MS	$299
QuickCode III Plus	MS	$239

Fortran & Supporting

50:More FORTRAN	PC	$ 99
ACS Time Series	MS	$399
Forlib+ by Alpha	MS	$ 59
MS Fortran - 4.0, full '77	MS	$279
No Limit - Fortran Scientific	PC	$115
PC-Fortran Tools - xref, pprint	PC	$165
RM/Fortran	MS	Call
Scientific Subroutines - Matrix	MS	$139

Multilanguage Support

BTRIEVE ISAM	MS	$185
BTRIEVE/N-multiuser	MS	$455
Flash-Up Windows	PC	$ 79
GSS Graphics Dev't Toolkit	PC	$375
HALO Graphics	PC	$205
Development Package	MS	$389
Informix 4GL-application builder	PC	$789
Informix SQL - ANSI standard	PC	$639
Opt Tech Sort - sort, merge	MS	$119
PANEL	MS	$215
Pfinish - by Phoenix	MS	$229
PolyLibrarian by Polytron	MS	$ 79
PolyBoost - speed I/O, keyboard	PC	$ 69
PVCS Corporate-source control	MS	$309
QMake by Quilt Co.	MS	$ 79
Report Option - for Xtrieve	MS	$109
Screen Machine	PC	$ 59
Screen Sculptor	PC	$ 95
SRMS - source control	MS	$109
Synergy - create user interfaces	MS	$375
VXM - multi-env. link	MS	$195
Xtrieve - organize database	MS	$199
ZAP Communications - VT 100	PC	$ 89

FEATURE

NET-TOOLS - Access NETBIOS-compatible network systems from Microsoft C, Pascal, FORTRAN, Assembler, Lattice C. Full Source, No Royalties. PC $129

We support MSDOS (not just compatibles), PCDOS, Xenix-86, CPM-80, Macintosh, Atari ST, and Amiga.

VISA **HOURS** MasterCard
8:30 AM-8:00 PM EST.

Call for a catalog, literature, advice and service you can trust

"I like the way you do business. I ordered two programming packages from you and before I had time to wonder when I would receive them, they were on my desk. Good prices and fast delivery, - not a bad way to do business. - Thanks"
Jeff Schropfer
Bytrek

CIRCLE 133 ON READER SERVICE CARD

THE PROGRAMMER'S SHOP

provides complete information, advice, guarantees and every product for Microcomputer Programming.

AUTO-INTELLIGENCE

The First Automatic Knowledge Acquisition System

List: $990 Our: $749

IntelligenceWare, Inc.

9800 S. Sepulveda Blvd. Suite 730
Los Angeles, CA 90045
Telephone: (213) 417-8896; Fax (213) 417-8897

CIRCLE 301 ON READER SERVICE CARD

WANT TO ADD

WINDOWS, ICONS, FONTS, FAST GRAPHICS, DIALOG BOXES, PROCESS MANAGEMENT, AND DEVICE INDEPENDENCE

TO YOUR IBM PC PROGRAMS?

If you have ever wished that you could develop stunning Macintosh-like programs on the IBM PC without the overhead of an enormous operating environment like Windows or GEM, then you need the **SYNERGY DEVELOPMENT TOOLKIT, from Matrix Software.**

Using a memory resident runtime module only **20K** in size (versus as much as 300K for Windows), you can develop applications with: overlapped and tiled windows; pull-down menus with half intensity options and automatic sizing; fast graphics function calls to draw shapes, lines, boxes, and create intricate fill patterns in both regular and irregular areas; have full device independence (drivers for numerous devices, including CGA, EGA and Hercules are included); sophisticated text input and output, with fonts in different styles and sizes; full keyboard support (your programs won't need a mouse!) and powerful mouse support; and process management calls to efficiently manipulate system resources.

The Toolkit has gateways to support the following languages:

- Turbo Pascal
- IBM/MS Pascal
- Microsoft & Lattice C
- Macro Assembler
- Basic
- dBASE II/III Compilers

In addition, the Toolkit includes a powerful collection of tools including a graphics resource editor for developing your own icons and fonts.

NEW! The Toolkit also **includes a free copy of Synergy Layout**, a revolutionary software development tool that dramatically increases your productivity by actually generating bug-free source code in Macro, C, and Turbo Pascal.

For further information, contact Matrix Software at (617) 567-0037.

List: $395
Our: $349

CIRCLE 302 ON READER SERVICE CARD

MUMPS

Now quickly and easily develop multi-tasking applications on your PC!

Now get the speed, power and flexibility of MUMPS in a truly concurrent processing environment with COMP Computing Standard MUMPS (CCSM).

CCSM provides the programmer with these benefits:

* Save time and money by writing Applications in 1/3 to 1/5 the amount of code.
* Fast data access with B-Tree File Structure and Automatic disk caching.
* Unlimited program size and variable space with advanced virtual memory system.
* 100% portable from micros to minis to mainframes.
* Easily write multi-tasking and multiuser applications.

And with MUMPS, all you need to do to start a background process is: "JOB ^ROUTINE". No memory allocation, no table set-up, no hassles; the system handles everything.

CCSM is also available in a multi-tasking version for the Macintosh.

Multi-Tasking:	List - $149.95	Ours - $129
Multiuser:	List - $450.00	Ours - $369

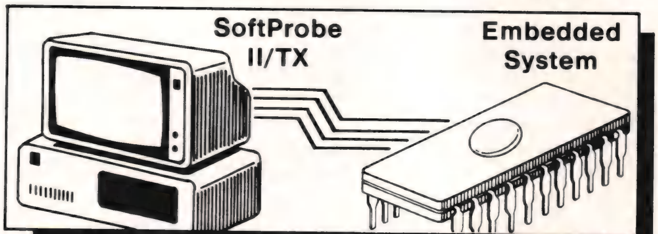
For a demonstration diskette of CCSM, send two dollars to the address below.

MGlobal
1601 Westheimer, Suite 201
Houston, Texas 77006

1-800-257-8052
In Texas: 1-713-529-2576

CIRCLE 303 ON READER SERVICE CARD

SoftProbe II/TX, a High-Level Solution for Embedded System Debugging.



FEATURES

- Source Level Debug
- Supports High Level Language Data Types
- C-like Command Language with Expression Evaluation and Flow Control
- Displays Program Execution Trace in High Level and Assembly Languages
- Friendly User Interface with Macros and On-line Help

TARGET SYSTEM

- Any iAPX-86/186 Based System with a USART
- IBM PC or Compatible.

BENEFITS

- Debug on Actual Target System
- Debug at High Level Language Level
- Mixed Language Debugging Facilitates Firmware Testing

LANGUAGE SUPPORT

- C
- PL/M
- ASM

SYSTEMS & SOFTWARE
3303 Harbor Blvd.,
C-11, Costa Mesa, CA 92626
(714) 241-8650 FAX (714) 241-0377

List: $750 Ours: $695

SoftProbe is a registered trademark of Systems & Software, Inc.
IBM PC is a registered trademark of International Business Machines Corp.

CIRCLE 304 ON READER SERVICE CARD

Call Today for FREE detailed information or try Risk-Free for 31 days.

800-421-8006
HOURS: 8:30 A.M. - 8:00 P.M. E.S.T.

THE PROGRAMMER'S SHOP™
Your complete source for software, services and answers
5-D Pond Park Road, Hingham, MA 02043
Mass: 800-442-8070 or 617-740-2510 2/87

Listing One *(Listing continued, text begins on page 94.)*

```
vb_replace(c)
{
        /* Overwrite the character at the current cursor
         * position without moving the cursor.
         */

        Regs.h.ah = 10 ;
        Regs.h.al = c;              /* write c              */
        Regs.h.bl = 0x07;           /* Normal characters    */
        Regs.h.bh = 0;              /* Display page 0       */
        Regs.x.cx = 1;              /* # of times to write  */

        int86( VIDEO_INT, &Regs, &Regs );
}

/*-------------------------------------------------------*/

vb_putchar( c )
{
        /* Write a character to the screen in TTY mode.
         * Only normal printing characters, BS, BEL, CR and
         * LF are recognized. The cursor is automatically
         * advanced and lines will wrap.
         */

        Regs.h.bl = 0x07;
        Regs.h.al = c;
        Regs.h.ah = WRITE_TTY ;
        int86( VIDEO_INT, &Regs, &Regs );
}

/*-------------------------------------------------------*/

vb_puts( str, move_cur )
register char   *str;
{
        /* Write a string to the screen in TTY mode. If
         * move_cur is true the cursor is left at the end
         * of string. If not the cursor will be restored to
         * its original position (before the write).
         */

        register int    posn;

        if( !move_cur )
                posn = vb_getcur();

        while( *str )
                vb_putchar( *str++ );

        if( !move_cur )
                vb_setcur( posn );
}

/*-------------------------------------------------------*/

int     vb_getchar()
{
        /* Get a character with a direct video bios call.
         * This routine can be used to complement stderr as
         * it can be used to get characters from the keyboard,
         * even when input redirected. The typed character
         * is returned in the low byte of the returned
         * integer, the high byte holds the auxillary byte
         * used to mark ALT keys and such. See the Technical
         * Reference for more info.
         */

        Regs.h.ah = 0 ;
        int86( KB_INT, &Regs, &Regs );
        return( (int)Regs.x.ax );
}

/*-------------------------------------------------------*/
#ifdef MAIN

main()
{
        vb_replace( 'X' );
        vb_putchar('\n');
        vb_putchar('\r');
}

#endif
```

End Listing One

(Listing Two begins on page 80.)

Listing Two

(Text begins on page 94.)

```
                    Listing 2 -- /include/box.h

/*----------------------------------------------------------*
 * BOX.H:  Copyright (c) 1987, Allen I. Holub.
 *                All rights reserved.
 *
 *   #defines for the box-drawing characters on the IBM PC.
 *
 *----------------------------------------------------------*
 *      The names are:
 *
 *      UL      Upper left corner
 *      UR      Upper right corner
 *      LL      lower left corner
 *      LR      lower right corner
 *      CEN     Center (intersection of two lines)
 *      TOP     Tee with the flat piece on top
 *      BOT     Bottom tee
 *      LEFT    Left tee
 *      RIGHT   Right tee
 *      HORIZ   Horizontal line
 *      VERT    Vertical line.
 *
 *
 *                      UL    -TOP-    UR            HORIZ
 *                              |
 *                      L               R               V
 *                      E       |       I               E
 *                      F--   -CEN-   --G               R
 *                      T       |       H               T
 *                      |               T
 *                              |
 *                      LL    -BOT-    LR
 *
 *
 * The D_XXX  defines have double horizontal and vertical lines
 * The HD_XXX defines have double horizontal lines only
 * The VD_XXX defines have double vertial lines only
 *
 * If your terminal is not IBM compatible, #define all of these
 * as '+', except for the VERT #defines, which should be a |,
 * and the HORIZ #defines, which should be a -.
 */

#define VERT            179
#define RIGHT           180
#define UR              191
#define LL              192
#define BOT             193
#define TOP             194
#define LEFT            195
#define HORIZ           196
#define CEN             197
#define LR              217
#define UL              218

#define D_VERT          186
#define D_RIGHT         185
#define D_UR            187
#define D_LL            200
#define D_BOT           202
#define D_TOP           203
#define D_LEFT          204
#define D_HORIZ         205
#define D_CEN           206
#define D_LR            188
#define D_UL            201

#define HD_VERT         179
#define HD_RIGHT        181
#define HD_UR           184
#define HD_LL           212
#define HD_BOT          207
#define HD_TOP          209
#define HD_LEFT         198
#define HD_HORIZ        205
#define HD_CEN          216
#define HD_LR           190
#define HD_UL           213

#define VD_VERT         186
#define VD_RIGHT        182
#define VD_UR           183
#define VD_LL           211
#define VD_BOT          208
#define VD_TOP          210
#define VD_LEFT         199
#define VD_HORIZ        196
#define VD_CEN          215
#define VD_LR           189
#define VD_UL           214
```

End Listing Two

Listing Three

```
                    Listing 3 -- /include/curses.h

/*----------------------------------------------------------*
 * CURSES.H:  Copyright (c) 1987, Allen I. Holub.          *
 *                All rights reserved.                      *
 *----------------------------------------------------------*
 */

typedef struct
{
    int x_org;      /* X coordinate of upper-left corner     */
```

```
    int y_org;     /* Y coordinate of upper-left corner     */
    int x_size;    /* Horizontal size of text area.         */
    int y_size;    /* Vertical    size of text area.        */
    int row;       /* Current cursor row (0 to y_size-1)    */
    int col;       /* Current cursor column (0 to x_size-1)*/
    int scroll_ok  /* Scrolling permitted in this window    */
}
WINDOW;

# define         bool     unsigned int
# define         reg      register
# define         TRUE     (1)
# define         FALSE    (0)
# define         ERR      (0)
# define         OK       (1)

/*----------------------------------------------------------------
 * Reminder: The comma opperator goes left to right and
 *               evaluates to the rightmost thing in the list.
 *
 * The following macros implement many of the curses functions
 * note that stdscr only has meaning when passed to getyx.
 *----------------------------------------------------------------
 */

#define stdscr          0

#define getyx(win,y,x)   (win ? ((x)=win->col,(y)=win->row)  \
                          : getpos(&x,&y))

#define mvinch(y,x)      ( move (  y,x), inch ( ) )
#define mvwinch(w,y,x)   ( wmove(w,y,x), winch(w) )

#define addch(c)              putchar(c)
#define endwin()
#define erase()               clear()
#define initscr()
#define printw                printf
#define refresh()
#define scroll(w)             wscroll(w,1);
#define scrollok(win,flag)  ((win)->scroll_ok = (flag))
#define subwin(w,a,b,c,d)   newwin(a,b,c,d)
#define wclear               werase
#define wrefresh(win)

/*-----------------------------------------------------------*/

extern  WINDOW *newwin  (int ,int ,int ,int );
extern  int box         (WINDOW *,int ,int );
extern  int clear       (void);
extern  int crmode      (void);
extern  int echo        (void);
extern  int getch       (void);
extern  int move        (int ,int );
extern  int nl          (void);
extern  int nocrmode    (void);
extern  int noecho      (void);
extern  int nonl        (void);
extern  int waddch      (WINDOW *, int  );
extern  int waddstr     (WINDOW *, char *);
extern  int wclrtoeol   (WINDOW *       );
extern  int werase      (WINDOW *       );
extern  int wgetch      (WINDOW *       );
extern  int wmove       (WINDOW *, int   ,int );
extern  int wprintw     (WINDOW *, char *,... );
extern  int wscroll     (WINDOW *, int       );
```

End Listing Three

Listing Four

```
                        Listing 4 -- curses.c

#include <stdio.h>
#include <ctype.h>
#include <curses.h>  /* routines in this file.              */
#include <box.h>     /* of IBM box-drawing characters       */
#include <stdarg.h>  /* va_list and va_start (ANSI)         */

/*----------------------------------------------------*
 * CURSES.C:  Copyright (c) 1987, Allen I. Holub.      *
 *                All rights reserved.                 *
 *----------------------------------------------------*
 *
 * This file is a DOS implementation of some of the Unix
 * CURSES functions. It is Unix compatible but is a proper
 * subset, not a full implementation, of curses. It works
 * on the IBM-PC. In all of these y is the row number and
 * x is the column number. The upper left corner is (0,0):
 *
 * I/O functions -------------------------------------------
 *
 * waddch( win, ch )        Works like putc()
 * waddstr( win, str)       Works like puts()
 * wprintw(win,fmt,arg...)  Like printf but writes to the
 *                          indicated window.
 * wrefresh(win)            See below.
 * box(win, vert, horiz)    Draws a box around the window.
 *
 * Cursor movement and screen control ----------------------
 *
 * werase(win)          erase the entire window
 * wclrtoeol(win)       erase from cursor position to the end
 *                      of line
 *
```

(continued on next page)

Listing Four

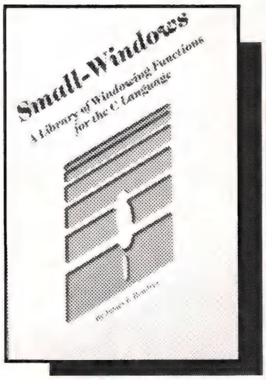
(Listing continued, text begins on page 94.)

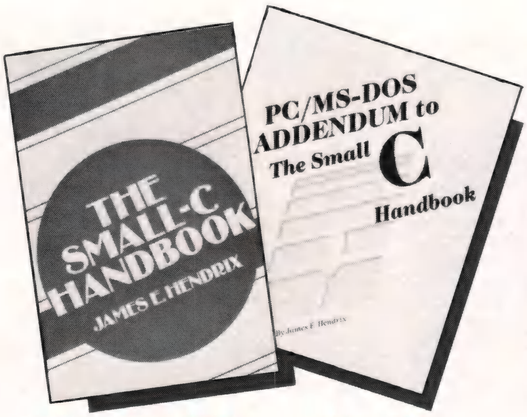

```
 * wmove(win,y,x)  Move the cursor to postion (y,x) rela-
 *                 tive to the orgin of the indicated window.
 *
 * getxy(win,y,x)      MACRO: puts the current cursor postion
 *                     into y and x. Note that this is a macro,
 *                     don't put an & in front of y and x in
 *                     the invocation.
 *
 * wgetch(win)         works like getchar but echos to the
 *                     indicated window.  If crmode is
 *                     inactive, it is activated for the
 *                     duration of this subroutine.
 *
 * scroll(win)              Scrolls the window up one line.
 * wscroll( win, amt )  NOT A CURSES FUNCTION. Scrolls
 *                          window by indicated amount. A
 *                          negaitve amt scrolls down.
 *
 * Initialization stuff ----------------------------------------
 *
 * initscr()                Initialize
 * endwin()                 Clean up
 * scrollok(win, flag);     enable/disable scrolling for window.
 * newwin(lines,cols,begin_y,begin_x)      Create a new window.
 *
 * Terminal control -------------------------------------------
 *         Because of the perversities of DOS, these work in a
 *         slightly nonstandard way. In particular echo,noecho,
 *         nl, and nonl only work if crmode() is active.
 *         Moreover, they are ignored for the non-w fundtions.
 *         For portability reasons, it's best to always set
 *         crmode() at the top of your program.
 *
 * crmode();                Turn off input buffering.
 * nocrmode();              Turn it back on again.  (default)
 * noecho();                Turn off automatic echo.
 * echo();                  Turn it back on again.  (default)
 * nl();                    Turn on CR-NL mapping   (default)
 * nonl();                  turn it off again.
 *
 * Functions that affect the whole screen. -----------------
 *
 * move(y,x)                move cursor to abs. position (y,x)
 * addch(c)                 Write a character.
 * clear()                  Clear the screen
 * printw()                 works like printf
 * getch()                  get a character from the keyboard.
 * refresh()                See below.
 *
 *-------------------------------------------------------------
 * The real curses keeps an two internal representations of
 * the screen, when you change something it just modifies
 * one of these reprsentations. You must issue a refresh()
 * or wrefresh() call to actually modify the screen. My
 * version of curses refreshes the screen immediately after
 * every write. refresh() and wrefresh() macros have been
 * provided for UNIX compatability, however. These macros
 * don't do anything, but you should scatter them liberally
 * about your code if you want it to be portable.
 *
 * I've corrected one bug in the real curses that might
 * cause problems when you port your code. The real curses
 * (at least the one at Berkeley), doesn't scroll properly
 * in that it leaves junk on the bottom line of the window
 * after a scroll. I've corrected the problem here but,
 * again, if you want real portability you should do a
 * wclrtoeol(win) after every scroll. Unfortunately, there's
 * no way to determine that the screen has scrolled without
 * actually keeping track of the characters that are written
 * to the screen. Ugh.
 *
 * Other differences: curses doesn't know about characters
 * that it hasn't actually put on the screen with an addch().
 * So, if echo is enabled, curses won't erase the echoed
 * characters when it scrolls the screen and so forth. The
 * curses presented here doesn't exibit this behaviour, but
 * if you want compatablility with Unix, turn off echo (with
 * a noecho() call) and then echo all characters yourself.
 *
 * The purpose of the four #defines immediately below is to
 * make it easy to modify this package. They all map function
 * calls to video-bios subroutines [vb_xxx()]. If you want to
 * use your own functions (that do direct video access or send
 * out escape sequences to a normal terminal, for example)
 * just change the #defines and recompile. The functions must
 * behave as follows:
 *
 * cmove(y,x)     Move the cursor to position (y,x) (y is row)
 *                where (0,0) is the upper-left corner of the
 *                screen.
 * curpos(y,x)    Put the (y,x) cursor position into the
 *                integers pointed to by y and x.
 * replace(c)     Print c at the current cursor position without
 *                moving the cursor. This routine must be able
 *                to put a character into the 80th column
 *                without scrolling the screen or wrapping
 *                around.
 *
 * doscroll(l,r,t,b,a)  Scroll a region of the screen with the
 *                      top left corner at (t,l) and the bottom
 *                right corner at (b,r). "a" is the number of
 *                lines to scroll (up if a is positive, down if
 *                it's negative).
 *
```

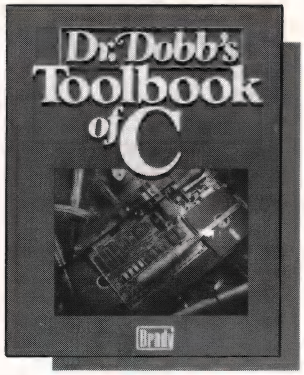

(continued on page 89)

M & T BOOKS AND SOFTWARE TOOLS

C Toolbox







Small-Windows: A Windowing Library

Small-Windows is a complete windowing library for C and Small-C. The package contains: 18 video functions written in assembly language; 7 menu functions that support both static and pop-up menus; and 41 window functions, including functions to clean, frame, move, hide, show, scroll, push, and pop windows. A file directory facility illustrates the use of the window menu functions and provides file selection, renaming and deletion capability. Two test programs are also included. For PC/MS-DOS systems, and Microsoft C Version 4.0 and Small-C compilers. Documentation and full source code is included.

Small-Windows Item #109 $29.95

Small-Tools: Programs for Text Processing

These Small-C programs perform specific, modular operations on text files, including: editing, formatting, sorting, merging, listing, printing, searching, changing, transliterating, copying, concatenating, encrypting and decrypting, and more. Supplied as source code. With the *Small-C Compiler* you can select and adapt these tools to meet your own needs. Documentation is included.

Small-Tools Item #010A $29.95

Small-C Compiler & Small-C Handbook

Like a home-study course in compiler design, *the Small-C Compiler* and the *Small-C Handbook* provide everything you need for learning how compilers are constructed, and for learning C at its most fundamental level. Full source code is included.

CP/M Small-C Compiler with Handbook
Item #006B $37.90

MS/PC-DOS Small-C Compiler with Handbook and Addendum
Item #006C $42.90

Small-Mac: An Assembler for Small-C

This assembler includes: a simplified macro facility, C language expression operators, object file visibility, descriptive error messages, and an externally defined instruction table. You'll receive the macro assembler, linkage editor, load-and-go loader, library manager, CPU configuration utility, and a utility to dump relocatable files. Documentation is included. For CP/M systems only.

Small-Mac Item #012A $29.95

C Disk Formats
Please indicate MS/PC-DOS or CP/M. For CP/M specify: Apple, Osborne, Kaypro, Zenith Z-100 DS/DD, 8" SS/SD.

Dr. Dobb's Toolbook of C

This authoritative reference contains over 700 pages of the best C articles and source code from *Dr. Dobb's Journal,* along with new material by C experts. The level is sophisticated and pragmatic, appropriate for professional C programmers. You'll find hundreds of pages of valuable C source code, including a complete compiler, an assembler, and text processing utilities.

Toolbook of C Item #005 $29.95

Special Packages

Save over $27!
Receive: Dr. Dobb's Toolbook of C, The Small-C Handbook and Small-C Compiler, Small-Mac Assembler, and Small-Tools Text Processing Programs. Only $99.95!

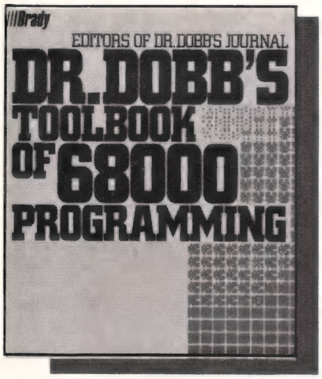
CP/M Package Item #005A $99.95

Save $22!
Receive: Dr. Dobb's Toolbook of C, The Small-C Handbook and MS/PC-DOS Addendum, Small-C Compiler, Small-Tools Text Processing Programs and Small Windows. Only $109.95! Specify Microsoft C Version 4.0 or Small-C compiler.

MS/PC-DOS Package
Item #005W $109.95

M & T BOOKS AND SOFTWARE TOOLS

Assembly Language Programming for the 68000 & Z80



Dr. Dobb's Toolbook of 68000 Programming

This complete collection of practical programming tips and techniques for the 68000 family includes the best articles on 68000 programming ever published in Dr. Dobb's, along with much new material.
Contents include:

An Introduction to the 68000 Family
- 68000 Instruction Set

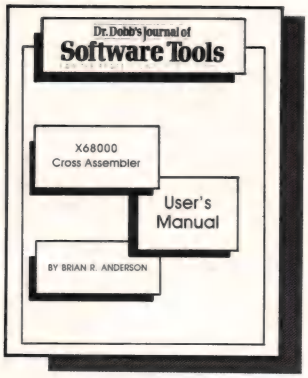
Development Tools
- Bringing Up the 68000: A First Step
- A 68000 Cross-Assembler

Useful 68000 Routines and Techniques
- A Simple Multitasking Kernel for Real-Time Applications
- The Worm Memory Test
- A Mandelbrot Program for the Macintosh

All programs are also available on disk!

68000 Toolbook	**Item #040**	**$29.95**
68000 Toolbook with disk	**Item #041**	**$49.95**

Specify MS-DOS, CP/M, CP/M 8", Osborne, Macintosh, Amiga, Atari 520st.

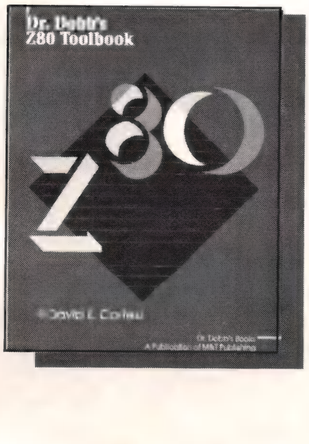


68000 Cross Assembler

An executable version of the 68000 Cross-Assembler discussed in the book is also available, complete with source code and documentation. Requires CP/M 2.2 with 64k or MS-DOS with 128k.

68000 Cross Assembler	**Item #042**	**$25.00**

Specify 8" SS/SD, Osborne, or MS-DOS.



Dr. Dobb's Z80 Toolbook

by David E. Cortesi

Dr. Dobb's Z80 Toolbook puts the power of assembly language in the hands of anyone who's done a little programming. You'll find:

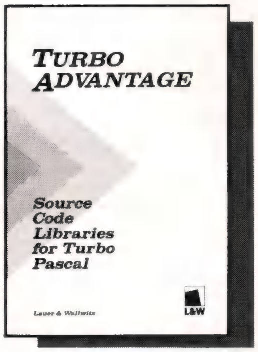
- A method of designing programs and coding them in assembly language and a demonstration of the method in the construction of several complete, useful programs.
- A complete, integrated toolkit of subroutines for arithmetic, string-handling, and total control of the CP/M file system.
- Every line of the toolkit's source code is there for you to read.

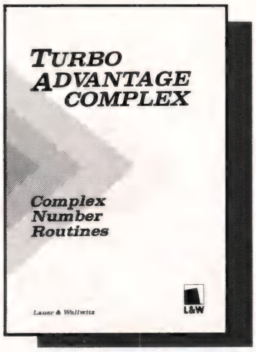
All the software—the programs plus the entire toolkit, both as source code and object modules for both CP/M 2.2 and CP/M Plus—is yours on disk. Most of the programs are included in the book, however, the disk is necessary for complete listings. A DRI RMAC assembler or equivalent is required.

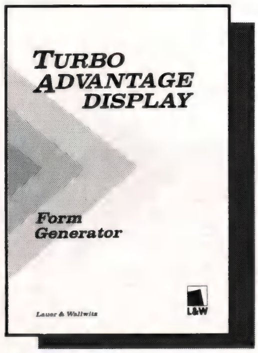
Dr. Dobb's Z80 Toolbook	**Item #022**	**$25**
Dr. Dobb's Z80 Toolbook with disk	**Item #022A**	**$40**

Specify 8" SS/SD, Apple, Osborne or Kaypro.

Turbo Pascal Tools







TURBO Advantage:

Source Code Library for Turbo Pascal

This library of more than 220 routines—complete with source code, sample programs and documentation—will save you hours of work developing and optimizing your programs!

Routines are organized and documented under the following categories: bit manipulation, file management, MS-DOS support, sorting, string operations, arithmetic calculations, data compression, differential equations, Fourier analysis and synthesis, matrices and vectors, statistics, and much more! All source code is included.

A detailed manual includes a description of the routine, an explanation of the methods used, the calling sequence, and a simple example. For MS/PC-DOS systems.

TURBO Advantage: Source Code Libraries for Turbo Pascal **is also available with** ***TURBO Advantage Complex: Complex Number Routines for Turbo Pascal*** **and** ***TURBO Advantage Display: Form Generator for Turbo Pascal.***

Turbo Advantage Item #070 $49.95

TURBO Advantage Complex:

Complex Number Routines for Turbo Pascal

Working with complex numbers is easy with the Turbo Pascal procedures and routines provided in TURBO Advantage Complex!

TURBO Complex provides procedures for performing all the arithmetic operations and necessary real functions with complex numbers. Each procedure is based on predefined constants and types. By using these declarations the size of arrays are easily adapted. Each type declaration is a record with both a real and imaginary part. Use these procedures to build more sophisticated functions in your own programs.

TURBO Complex also demonstrates the usage of these procedures in routines for vector and matrix calculation with complex numbers and variables; simultaneous Fourier transforms; calculations of convolution and correlation functions; low-pass, high-pass, band- pass and band-rejection digital filters; and in solving linear boundary-value problems.

Source code and documentation is included for MS-DOS systems. Some of the *TURBO Complex* routines are most effectively used with routines contained in *TURBO Advantage.*

TURBO Advantage/ Complex Package	Item #070A	$115
TURBO Complex	Item #071	$89.95

TURBO Advantage Display:

Form Generator for Turbo Pascal

Now, even if you have little programming knowledge, you can design and process forms to fit your needs!

TURBO Display includes a menu-driven form processor, 30 Turbo Pascal procedures and functions to facilitate linking created forms to your program, and full source code and documentation. For MS-DOS systems.

Some of the *TURBO Advantage: Source Code Libraries for Turbo Pascal* routines are necessary to compile *TURBO Display.* You save $20 when you order *TURBO Advantage: Source Code Libraries for Turbo Pascal* together with *TURBO Display: Form Generator for Turbo Pascal.* Receive both for only $99.95!

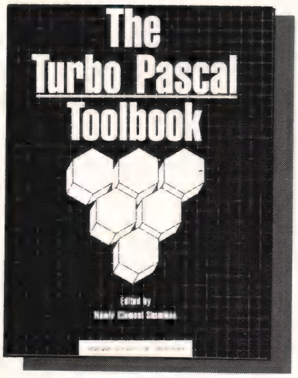
TURBO Display: Form Generator for Turbo Pascal is also available individually for $69.95.

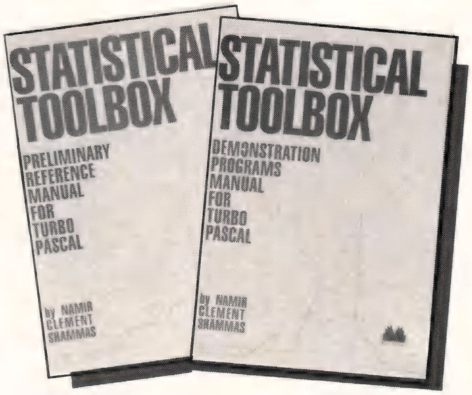
TURBO Advantage/ Display Package	Item #070B	$99.95
TURBO Display	Item #072	$69.95

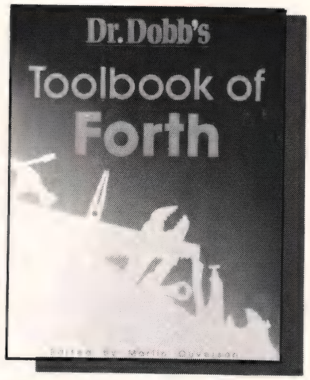
Each Turbo Advantage package includes complete source code on disk.

M & T BOOKS AND SOFTWARE TOOLS

Programming Eloquence







The Turbo Pascal Toolbook

Edited by Namir Clement Shammas

Make your programming easier and more powerful with *The Turbo Pascal Toolbook!*

You'll find:

- an extensive library of low-level routines
- external sorting and searching tools, presenting a new database routine that combines the best features of the B-tree, B+ and B++ trees
- window management, to help you create, sort, and overlay windows
- artificial intelligence techniques
- mathematical expression parsers, offering two routines that convert mathematical expressions into RPN tokens
- a smart statistical regression model that searches for the best regression model to represent a given set of data.

All routine libraries and sample programs are on disk for MS-DOS systems, and over 800K of Turbo Pascal source code is included!

Turbo Pascal Toolbook
Item #080 **$25.95**
Turbo Pascal Toolbook with disk
Item #081 **$45.95**

STAT Toolbox

for Turbo Pascal

Bring convenience, power, and versatility to your statistics programs!

Two statistical packages in one!

A library disk and reference manual
Use these powerful statistical routines to build your applications.
Routines include:

- statistical distribution functions
- random-number generation
- basic descriptive statistics
- parametric and non-parametric statistical testing
- bivariate linear regression, multiple and polynomial regression.

A demonstration disk and manual
This package incorporates the library of routines into a fully functioning statistical program. Two data management programs are included to facilitate the storage and maintenance of data.

Full source code is included. (For IBM PC's and compatibles. Turbo Pascal version 2.0 or later, and PC DOS 2.0 or later are required).

STAT Toolbox **Item #050** **$69.95**

Dr. Dobb's Toolbook of Forth

This comprehensive collection of useful Forth programs and tutorials contains *DDJ's* best Forth articles, expanded and revised along with new material. You'll find sections on:

- Mathematics in Forth
- Modifications/Extensions
- Forth Programs
- Forth—the Language
- Implementing Forth

You'll also find Appendixes that will help you convert fig Forth to Forth-83, and tell you how to stay up-to-date on the latest developments and refinements of this popular language.

The screens in the book are also available on disk as ASCII files.

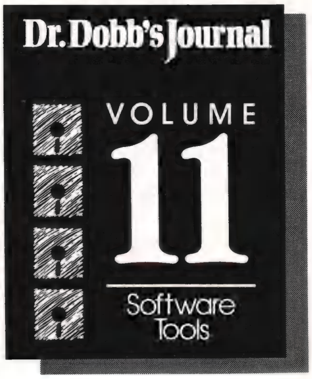
Dr. Dobb's Toolbook of Forth
Item #030 **$22.95**
Dr. Dobb's Toolbook of Forth with Disk
Item #031 **$39.95**

Please specify MS/PC-DOS, Apple II, Macintosh, or CP/M. For CP/M disks, specify Osborne or 8" SS/SD.

CALL
800-533-4372
TOLL FREE

In CA 800-356-2002

Comprehensive Reference Guides



SPECIAL OFFER

Receive 11 years' worth of useful code and fascinating history—the complete Dr. Dobb's library of 11 volumes for only $299. That's a savings of over $60!
Item #020A **$299**

Dr. Dobb's Bound Volume 11

All of 1986!

Bound Volume 11: 1986
The promise of power. With the introduction of the first true 32-bit microprocessors, desktop computers began to rival minis and mainframes in power. *DDJ* covered the changes with special issues on the 68000, parallel processing, artificial intelligence, the 80386, and multitasking. We supported the new chips with assemblers, translators, and other cross-development tools. Additional features included a special graphics issue, reviews of Dan Briklin's DEMO program and Jef Raskin's SwyftCard, and our sixth annual Forth issue. We introduced a new structured languages column, led by Michael Ham and Namir Shammas, while Ray Duncan and Allen Holub continued to provide their own valuable columns.
Item #020F **$35.75**

Bound Volume 1: 1976
The working notes of a technological revolution. Before there was Apple, *DDJ* put a programming language on the first microcomputers and became a chronicler and instrument of the microcomputer revolution.
Item #013 **$30.75**

Bound Volume 2: 1977
Running light without overbyte. By year two the formula was clear: serious technical questions handled with a minimum of reverence much source code and a commitment to tight coding.
Item #014 **$30.75**

Bound Volume 3: 1978
The roots of Silicon Valley growth. The S-100 bus was hashed out in *DDJ*'s pages. Steve Wozniak and others published in *DDJ* code would help to build an industry.
Item #015 **$30.75**

Bound Volume 4: 1979
In the midst of the gold rush. Three years before IBM moved in, the neighborhood was less civilized. *DDJ* published a gold mine of tips, tricks, and algorithms.
Item #016 **$30.75**

Bound Volume 5: 1980
C and CP/M. 1980 saw an all-CP/M issue, including Gary Kildall's history of CP/M and Ron Cain's original Small-C Compiler.
Item #017 **$30.75**

Bound Volume 6: 1981
The first of Forth. This was the year *DDJ* launched its first Forth issue and Dr. Dobb's Clinic. Plus: PCNET, the Conference Tree, and 6809 Tiny BASIC.
Item #018 **$30.75**

Bound Volume 7: 1982
Legitimacy. *DDJ* Observed the IBM phenomenon, reviewed MS-DOS and CP/M-86, and looked forward to fifth-generation computers.
Item #019 **$35.75**

Bound Volume 8: 1983
Power tools. Professional software development on a PC was getting easier; *DDJ* helped with Small-C, the RED editor, and an Ada subset.
Item #020 **$35.75**

Bound Volume 9: 1984
Shaping things to come. In 1984 *DDJ* examined new programming environments: Prolog, expert systems, Modula-2, and a $49.95 Pascal. Plus Allen Holub's GREP, Unix internals, and two encryption systems.
Item #020B **$35.75**

Bound Volume 10: 1985
The year of living dangerously. In 1985, iconoclastic *DDJ* beat Apple to the goal of adding more memory, a SCSI port, and a hard disk to the Macintosh. Also: powerful software tools in C, Modula-2, Forth, Pascal, assembly language, and Prolog.
Item #020D **$35.75**

CALL
800-533-4372
TOLL FREE

In CA 800-356-2002

Order Form

ORDER NOW! ORDER NOW! ORDER NOW! ORDER NOW! ORDER NOW! ORDER NOW!

NAME ____________________

(Please use street address, not P.O. Box)

ADDRESS ____________________

CITY ____________ STATE ________ ZIP ________

DAY PHONE ____________________

Return form

OR

CALL

800-533-4372

TOLL FREE

In CA call

800-356-2002

☐ Check enclosed. Make payable to:

M&T Publishing, Inc., 501 Galveston Dr., Redwood City, CA 94063

☐ Charge My:

☐ Visa ☐ MasterCard ☐ American Express.

Name on card ____________________

Account No. ____________ Expiration Date ________

Signature ____________________

For disk orders, please indicate format. Refer to ad for standard format availability for each product.

☐ **MS/DOS** ☐ **Amiga** **CP/M**

☐ **Macintosh** ☐ **Atari 520st** ☐ Kaypro ☐ Osborne

☐ **Apple II** ☐ 8″ SS/SD ☐ Apple

☐ Zenith Z-100 DS/DD

For Small-Windows, indicate:

☐ Microsoft C version 4.0 compiler

☐ Small-C Compiler

QUANTITY	ITEM #	DESCRIPTION	UNIT PRICE	TOTAL PRICE

SUB-TOTAL ________

CA residents must add applicable sale tax on merchandise total ________ % SALES TAX ________

Shipping must be included with order. See rates below. SHIPPING ________

TOTAL ORDER ________

In U.S. For Bound Volumes, add $3.25 per book. Add $9.25 for Special C Packages. For other books and disks, add $2.25 per item. *Outside U.S.* For Bound Volumes, add $6.25 per book surface mail. Add $18 surface mail for Special C Packages. For other books and disks, add $5.25 per item surface mail. Foreign airmail rates available on request. For Fast Service and reduced shipping costs, Germans may order direct from: Markt & Technik, Buchverlag, Ilans-Pinsel-Strasse 2, 8013 Haar bei München. Call Germany 089-4613-383 or 089-4613-711 for prices in Deutsch Marks.

3129

M&T Books & Software

ORDER FORM

Please fold along fold-line and staple or tape closed.

No Postage
Necessary
If Mailed
In The
United States

BUSINESS REPLY MAIL

First Class Permit No. 871 Redwood City, CA

Postage Will Be Paid By Addressee

M&T Books & Software Tools
501 GALVESTON DRIVE
REDWOOD CITY, CA 94063

Please fold along fold-line and staple or tape closed.

Listing Four

(Listing continued, text begins on page 94.)

```
 * inchar()      Returns the character at the current curser
 *               position.
 */

#define cmove(y,x)              vb_ctoyx(y,x);
#define curpos(y,x)             vb_getyx(y,x);
#define replace(c)              vb_replace(c);
#define doscroll(l,r,t,b,a)     vb_scroll(l,r,t,b,a);
#define inchar()                vb_inchar(NULL)

#define min(a,b)                ((a) < (b) ? (a) : (b))

/*----------------------------------------------------------*/

static int Echo   = 1;  /* Echo enabled                     */
static int Crmode = 0;  /* If 1, use buffered input         */
static int Nl     = 1;  /* If 1, map \r to \n on input      */
                        /* and map both to \n\r on output */

/*----------------------------------------------------------*/

noecho   (      ) {  Echo = 0;                   }
echo     (      ) {  Echo = 1;                   }
nl       (      ) {  Nl   = 1;                   }
nonl     (      ) {  Nl   = 0;                   }
getpos   (yp,xp)  {  return curpos(yp, xp);      }
move     (y, x )  {  cmove(y, x);                }
inch     (      ) {  return inchar();            }

 /*---------------------------------------------------------*/

crmode()
{
     FILE        *console;

     setvbuf(stdin, NULL, _IONBF, 0);  /* Turn off buffering*/
     Crmode = 1;
}

nocrmode()
{
     freopen( "/dev/con", "r", stdin );
     Crmode = 0;
}

/*----------------------------------------------------------*/

WINDOW  *newwin( lines, cols, begin_y, begin_x )

int      cols;     /* Horizontal size (including border)   */
int      lines;    /* Vergtical size  (including bofder)   */
int      begin_y;  /* X coordinate of upper-left corner    */
int      begin_x;  /* Y coordinate of upper-left corner    */
{
         WINDOW  *win, *malloc();

         if( !(win = malloc( sizeof(WINDOW) )) )
                 ferr("Out of memory\n");

         win->x_org       = begin_x;
         win->y_org       = begin_y;
         win->x_size      = cols   ;
         win->y_size      = lines  ;
         win->row         = 0;
         win->col         = 0;
         win->scroll_ok   = 0;

         werase(win);
         return win;
}

/*----------------------------------------------------------*/

int      waddch( win, c )
WINDOW   *win;
int      c;
{
     /*  Print a character: The following are handled
      *  specially:
      *
      *  \n  Clear the line from the current cursor position
      *      to the right edge of the window. Then:
      *      if nl() is active:
      *          go to the left edge of the next line
      *      else
      *          go to the current column on the next line
      *      In addition, if scrolling is enabled, the window
      *      scrolls if you're on the bottom line.
      *  \t  is expanded into an 8-space field. If the tab
      *      goes past the right edge of the window, the
      *      cursor wraps to the next line.
      *  \r  gets you to the beginning of the current line.
      *
      *  \b  backs up one space but may not back up past
      *      the left edge of the window. Nondestructive. The
      *      curses documentation doesn't say that \b is
      *      handled explicitly but it does indeed work.
      *
      *  The following is not supported by Unix. Don't use
      *  explicit escape sequences if portability is a
      *  consideration:
      *
      *  ESC This is not standard curses but is useful. All
      *      characters between an ASCII ESC and an alphabetic
      *      character are sent to the output but are otherwise
```

(continued on next page)

Listing Four *(Listing continued, text begins on page 94.)*

```
     *       ignonred. This let's you send escape sequences
     *       directly to the terminal if you like. I'm
     *       assuming here that you won't change windows in the
     *       middle of an escape sequence.
     *
     * Return ERR if the character would have caused the
     * window to scroll illegally.
     */

    static  int      saw_esc = 0;
    static  WINDOW   *oldwin = NULL;
    int              rval    = OK;

    if( oldwin != win )
    {
        cmove(win->y_org + win->row, win->x_org + win->col);
        oldwin = win;
    }

    if( saw_esc )
    {
            if( isalpha( c ) )
                    saw_esc = 0;

            putchar(c);
    }
    else
    {
            switch( c )
            {
            case '\033':
                    saw_esc = 1;
                    putchar('\033');
                    break;

            case '\b':
                    if( win->col > 0 )
                    {
                            putchar('\b');
                            --( win->col );
                    }
                    break;

            case '\t':
                    do
                    {
                            waddch( win, ' ' );
                    }
                    while( win->col % 8 );
                    break;

            case '\r':
                    win->col = 0;
                    cmove(win->y_org + win->row, win->x_org);
                    break;

            default :
                    putchar(c);
                    if( ++(win->col) < win->x_size )
                            break;

                    /* fall through to newline */

            case '\n':
                    wclrtoeol( win );

                    if( Nl )
                        win->col = 0;

                    if( ++(win->row) >= win->y_size )
                    {
                        rval = wscroll( win, 1 );
                        --( win->row );
                    }

                    cmove( win->y_org + win->row,
                           win->x_org + win->col);
                    break;
            }
    }
```

```
    return rval;
}

/*-------------------------------------------------------*/

waddstr( win, str )
WINDOW  *win;
char    *str;
{
        while( *str )
                waddch( win, *str );
}

/*-------------------------------------------------------*/

static int Errcode = OK;

static wputc(c, win) WINDOW *win;
{
        Errcode |= waddch(win,c);
}

wprintw(win, fmt)
WINDOW  *win;
char    *fmt;
{
    /* The doprnt() function used here is explained in depth
     * in Allen Holub, The C Companion (Prentice-Hall: 1987),
     * pp. 213-237. If you don't have this book, an alternate
     * approach is explained in the text.
     */

    va_list args;
    va_start( args, fmt );

    Errcode = OK;
    doprnt( wputc, win, fmt, args );
    return Errcode;
}

/*-------------------------------------------------------*/

box( win, vert, horiz )
WINDOW  *win;
{
    /* Draws a box in the outermost characters of the window
     * using vert for the vertical characters and horiz for
     * the horizontal ones. I've extended this function to
     * support the IBM box-drawing characters. That is,
     * if IBM box-drawing characters are specified for vert
     * and horiz, box() will use the correct box-drawing
     * characters in the corners. These are defined in
     * box.h as:
     *
     *      HORIZ   (0xc4)  single horizontal line
     *      D_HORIZ (0xcd)  double horizontal line.
     *      VERT    (0xb3)  single vertical line
     *      D_VERT  (0xba)  double vertical line.
     */

    int     i, nrows;
    int     ul, ur, ll, lr;

    if( !( (horiz == HORIZ || horiz == D_HORIZ) &&
          (vert  == VERT  || vert  == D_VERT ) ))
    {
            ul = ur = ll = lr = vert ;
    }
    else
    {
        if( vert == VERT )
        {
            if(horiz==HORIZ)
                ul=UL,    ur=UR,    ll=LL,    lr=LR;
            else
                ul=HD_UL, ur=HD_UR, ll=HD_LL, lr=HD_LR;
        }
        else
        {
            if(horiz==HORIZ)
                ul=VD_UL, ur=VD_UR, ll=VD_LL, lr=VD_LR;
            else
                ul=D_UL,  ur=D_UR,  ll=D_LL,  lr=D_LR;
        }
    }

    cmove( win->y_org, win->x_org );

    putchar( ul );                              /* Top line */

    for( i = win->x_size-2; --i >= 0 ; )
            putchar( horiz );

    putchar( ur );
    nrows = win->y_size - 2 ;                   /* Two sides */
    i     = win->y_org  + 1 ;

    while( --nrows >= 0 )
    {
```

(continued on next page)

Listing Four

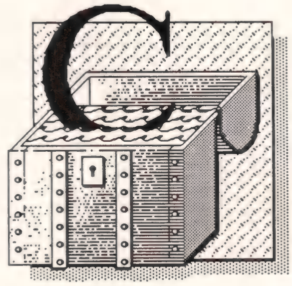
(Listing continued, text begins on page 94.)

```
            cmove( i,   win->x_org );
            putchar( vert );

            cmove( i++, win->x_org + (win->x_size - 1) );
            putchar( vert );
    }

    cmove(i, win->x_org);                   /* Bottom line */
    putchar( ll );

    for( i = win->x_size-2; --i >= 0 ; )
            putchar( horiz );

    putchar( lr );
}

/*-------------------------------------------------------*/

werase( win )
WINDOW  *win;
{
    vb_scroll( win->x_org, win->x_org + (win->x_size - 1),
               win->y_org, win->y_org + (win->y_size - 1),
               win->y_size );

    cmove( win->y_org, win->x_org );
    win->row = 0;
    win->col = 0;
}

/*--------------------------------------------------------
 * Scroll the window if scrolling is enabled. Return 1 if we
 * scrolled. (I'm not sure if the Unix function returns 1
 * on a scroll but it's convenient to do it here. Don't
 * assume anything about the return value if you're porting
 * to Unix. Wscroll() is not a curses function. It lets you
 * specify a scroll amount and direction (scroll down by -amt
 * if amt is negative); scroll() is a macro that evaluates to
 * a wscroll call with an amt of 1. Note that the Unix curses
 * gets very confused when you scroll explicitly (using
 * scroll()). In particular, it doesn't clear the bottom line
 * after a scroll but it thinks that it has. Therefore, when
 * you try to clear the bottom line, it thinks that there's
 * nothing there to clear and ignores your wclrtoeol()
 * commands. Same thing happens when you try to print spaces
 * to the bottom line; it thinks that spaces are already there
 * and does nothing. You have to fill the bottom line with
 * non-space characters of some sort, and then erase it.
 */

wscroll(win, amt)
WINDOW  *win;
{
     if( win->scroll_ok )
         doscroll( win->x_org, win->x_org + (win->x_size-1),
                   win->y_org, win->y_org + (win->y_size-1),
                   amt);

     return win->scroll_ok ;
}

/*-------------------------------------------------------*/

wmove( win, y, x )
WINDOW  *win;
{
     /* Seek into the window. It's not permitted to seek
      * outside of the window area.
      */

     cmove( win->y_org + (win->row = min(y,win->y_size-1)) ,
            win->x_org + (win->col = min(x,win->x_size-1)) );
}

/*-------------------------------------------------------*/

static  getcon( win )
WINDOW  *win;
{
     /* Get a character from DOS without echoing. We need
      * to do this in order to support (echo/noecho). We'll
      * also do noncrmode input buffering here. Maximum input
      * line length is 132 columns.
      *
      * In nocrmode(), DOS is used to get a line and all the
      * normal command-line editing functions are available.
      * Note that since there's no way to turn off echo in
      * this case, characters will be echoed to the screen
      * regardles of the status of echo().
      * In order to retain control of the window, input
      * fetched for wgetch() is always done in crmode, even
      * if Crmode isn't set.
      * If nl() mode is enabled, carriage return (Enter, ^M)
      * and linefeed (^J) are both mapped to '\n', otherwise
      * they are not mapped.
      */

     static unsigned char  buf[ 133 ] = { 133, 0 };
     static unsigned char  *p         = buf;
     static          int   numchars   = 0;
     register        int   c;

     if( Crmode || win )
     {
             if( (c = bdos(8,0,0) & 0xff) == ('Z'-'@') )
                     return EOF ;

             if( c == '\r' &&  Nl)
```

```
                c = '\n' ;

            if( Echo )
            {
                if( win )  waddch ( win, c );
                else       addch  (      c );
            }

            return c;
    }
    else if( numchars )
    {
            --numchars  ;
            return *p++ ;
    }
    else
    {
            bdos(10, buf, 0);
            numchars =  buf[1];
            p        = &buf[2];
    }
}

wgetch (win)  WINDOW *win; { return getcon( win  ); }
getch  (   )               { return getcon( NULL ); }

/*----------------------------------------------------------*/

clear()
{
        doscroll( 0, 79, 0, 24, 25 );
        move( 0, 0 );
}

/*----------------------------------------------------------*/

wclrtoeol( win )
WINDOW  *win;
{
    /* Clear from cursor to end of line, the cursor isn't
     * moved. The main reson that this is included here is
     * because you have to call it after printing every
     * newline in order to compensate for a bug in the real
     * curses. This bug has been corrected in the curses
     * presented here, however, so you don't have to use
     * this routine if you're not interested in portability.
     * Note that you must use a replace function on the
     * rightmost character to prevent scrolling.
     */

    register int    i;

    for( i = win->x_size - win->col - 1; --i >= 0 ;)
            putchar(' ');

    replace(' ');
    cmove( win->y_org + win->row, win->x_org + win->col )
}

/*----------------------------------------------------------*/
#ifdef MAIN

WINDOW  *boxwin( lines, cols, y_start, x_start )
{
    /* This routine works just like the newwin() except that
     * the window has a box around it that won't be destoyed
     * by writes to the window. It accomplishes this feat by
     * creating two windows, one inside the other, with a box
     * drawn around the outer one.
     */

    WINDOW *outer, *inner;

    outer = newwin(lines,   cols,   y_start,   x_start);

#ifdef MSDOS
    box( outer, VERT, HORIZ );
#else
    box( outer,  '|',   '-'  );
#endif

    wrefresh ( outer );
    return newwin( lines-2, cols-2, y_start+1, x_start+1 );
}

pattern()
{
    clear();
    printf("01234567890123456789012345678901234567890 0\n");
    printf("01234567890123456789012345678901234567890 1\n");
    printf("01234567890123456789012345678901234567890 2\n");
    printf("01234567890123456789012345678901234567890 3\n");
    printf("01234567890123456789012345678901234567890 4\n");
    printf("01234567890123456789012345678901234567890 5\n");
    printf("01234567890123456789012345678901234567890 6\n");
    printf("01234567890123456789012345678901234567890 7\n");
    printf("01234567890123456789012345678901234567890 8\n");
    printf("01234567890123456789012345678901234567890 9\n");
    printf("01234567890123456789012345678901234567890 10\n");
    printf("01234567890123456789012345678901234567890 11\n");
    printf("01234567890123456789012345678901234567890 12\n");
    printf("01234567890123456789012345678901234567890 13\n");
    printf("01234567890123456789012345678901234567890 14\n");
    printf("01234567890123456789012345678901234567890 15\n");
    printf("01234567890123456789012345678901234567890 16\n");
    printf("01234567890123456789012345678901234567890 17\n");
    printf("01234567890123456789012345678901234567890 18\n");
    printf("01234567890123456789012345678901234567890 19\n");
    printf("01234567890123456789012345678901234567890 20\n");
    printf("01234567890123456789012345678901234567890 21\n");
    printf("01234567890123456789012345678901234567890 22\n");
    printf("          1         2         3         4 n");
}

main()
{
        /* All coordinates are (y,x) */

        WINDOW  *win1, *win2;
        char    str[128];
        int     c;

        initscr(); /* Initialize curses                      */
        noecho();  /* Echo off (it screws up the screen) */
        crmode();  /* Put terminal into CBREAK mode          */

        pattern();

        win1 = boxwin(10, 20,  0, 0 );
        win2 = boxwin(10, 20, 21, 11);

        scrollok( win1, TRUE );
        wprintw(win1, "This is window one, doo wha\n" );
        wrefresh( win1 );
        wprintw(win2, "This is window 2.\nPress a key\n" );
        wrefresh( win2 );

        c = wgetch(win2);
        wmove(win1, 5, 0 );

        wprintw   (win1, "Got %c, 0x%x\n", c, c );
        wrefresh(win1);

        while( (c = wgetch(win1) & 0x7f) != 'q' )
        {
                if( c == 'X'-'@' )
                    wclrtoeol( win1 );
                else
                    waddch( win1, c );

                wrefresh( win1 );
        }

        move(23,0);
        refresh();
        endwin();
}
#endif
```

End Listings

Curses: Unix-Compatible Windowing Output Functions



As I've mentioned before, I write a lot of code that has to work in both the MS-DOS and Unix environments. Because I use a very Unix-compatible compiler (Microsoft 4.0), porting a program is usually pretty easy. Nonetheless, incompatibilities occasionally arise, usually when I'm doing some sort of low-level I/O. The Microsoft compiler doesn't have Unix-compatible *fcntl( )* or *ioctl( )* functions, it doesn't support the /dev/tty device for the console (you have to use /dev/con or con:), and it doesn't provide any sort of termcap- or curses-compatible function library. Of these, the most serious omission is the lack of a curses library.

For the uninitiated, curses is a collection of terminal-independent, low-level I/O functions. These subroutines let you do things such as move the cursor around on the screen, create and delete windows, write text and seek to specific window-relative cursor positions, and so forth. The windows can be overlapping, and they support individual wraparound, scrolling, and so on. The curses functions can talk to virtually any terminal. They accomplish this feat by using the termcap terminal database, which contains definitions for the various escape sequences needed to get around on specific terminals. Moreover, they talk to the terminals efficiently. That is, they always send out the minimum amount of characters necessary to modify the current screen.

by Allen Holub

Curses functions keep two internal images of the screen—one of these reflects what's actually on the screen; the other is a scratch space that you modify using the various curses functions. When you tell the curses functions to do a refresh, they compare the scratch buffer with the actual screen image and then send out the minimum number of characters necessary to get these images to match. This behavior is especially important when you're running a program via a modem and characters are coming at 1,200 baud. Redrawing the entire screen every time you scroll a 4-line by 10-character wide window is just unacceptable behavior. It takes too long. Curses functions solve the problem by redrawing only those parts of the screen that have actually changed. Curses functions are described in depth in Volume 2 of the *Unix Programmer's Manual* (see the bibliography).

This month I'm going to look at a set of curses-compatible I/O functions that run on the IBM PC. I've written several complex programs using these functions—programs that maintain several windows on-screen simultaneously, all of which are being updated at different rates. Moreover, the finished programs have ported to Unix with literally no modification. I have not implemented the entire curses library, however. My version of the curses package doesn't let you delete windows, nor does it support overlapping windows. Finally, it is bolted into the IBM PC. I've made no attempt to do the write optimizations discussed earlier, and I use several of the video BIOS functions to do things such as scroll the screen. If your terminal supports regional scrolling, however, it shouldn't be too difficult to modify the lowest-level scrolling function to send the proper escape sequences. Nevertheless, the package does give you quite a bit of DOS-to-Unix compatibility and has proved adequate for my own needs.

Interfacing to the IBM PC

All the curses functions talk to the screen via a well-defined video BIOS interface. I chose this approach for two reasons: first, the BIOS is somewhat faster than the normal DOS interface and is dramatically faster than the ANSI.SYS driver; and second, by concentrating the lowest-level routines in one place, I've (hopefully) made it easy to adapt these functions to terminals. The one problem that's likely to arise when porting this code to a terminal is with scrolling. I use the BIOS scroll function to scroll individual regions of the screen. You pass this function the coordinates of two diagonal corners of a square region on the screen and a scroll amount. The BIOS then scrolls only that region by the indicated amount. If your terminal doesn't have an escape sequence that does this, you'll have to do what the real curses package actually does—keep an internal image of the screen and refresh selected portions of it as part of the scroll. You could also do a series of character-read, move-cursor, character-write escape sequences, but that approach would be pretty slow.

The BIOS routines are all in vbios.c (Listing One, page 74). These subroutines are a reworking of the routines used by my shell (described in *On Command*, see the bibliography). They're sufficiently different from the original subroutines that I've reprinted them here. The actual mechanics of using the BIOS are discussed extensively by both Ray Duncan and Peter Norton (see the bibliography), so I won't go into the mechanics here. The supported functions are shown in Table 1, page 95.

The only other IBM-related file is box.h (Listing Two, page 80), which

holds *#defines* for the various IBM PC box-drawing graphics characters.

Curses

The curses package itself is part macros and part subroutines. The file curses.h (Listing Three, page 80) should be *#included* at the top of every file that uses the curses functions.

Supported functions are listed below, grouped functionally.

Initializing

Initialization functions are:

```
initscr()
endwin()
```

Initscr( ) initializes the curses package. It should be called at the head of your *main( )* subroutine, before any other curses functions are called. *Endwin( )* cleans up. It should always be called before your program exits.

In Unix programs, the terminal can be left in an unknown state if you abort your program with a Break. If you exit abnormally from a program that uses curses, only to find your terminal acting funny (not echoing, not handling tabs or new lines properly, and so forth), you can usually correct the problem by typing *tset* with no arguments. If that doesn't work, try *<NL>reset<NL>* where *<NL>* is a new line or Ctrl-J. If that doesn't work try *stty cooked echo nl*, and if that doesn't work, hang up and log on again. To avoid this sort of flailing around, it's much better for your program to trap the *SIGINT* signal and to call *endwin( )* from within the service subroutine. Use the following:

```
#include <signal.h>

onintr()
{
  endwin();
  exit(1);
}

main()
{
  signal( SIGINT, onintr );
}
```

None of the foregoing is a problem if you're not interested in porting your code to Unix, however.

Responding to Typed Characters

Once the curses package is initialized, you should determine how your terminal is going to respond to typed characters. Six subroutines are supported.

Two subroutines control input buffering:

```
int crmode ();
int nocrmode ();
```

Crmode( ) disables buffering. Characters will be available as soon as they're typed. A *nocrmode( )* call cancels a previous *crmode( )*. Here, an entire line is read before the first character is returned. The DOS (or Unix) command-line editing functions are all available if *nocrmode( )* is active. Most curses programs use *crmode( )*.

The following two subroutines control character echo:

```
int echo ();
int noecho ();
```

If *echo( )* is called, characters are echoed as they're typed; *noecho( )* suppresses the echoing, so you'll have to do the echo yourself. The real curses package gets very confused when *echo( )* is enabled. The problem here is that the curses package doesn't know about any character that it has not written to the screen itself. Because characters are echoed by the operating system (not by curses), the package doesn't know they're there. As a consequence, when the curses package does a screen refresh, it won't delete the characters that it doesn't know about and the screen rapidly fills with unwanted and unerasable

`int vb_getpage ( )`	Get active video page #.
`void vb_putchar (c)` `        int c;`	Write a single character to the screen at the current cursor position. Only printing characters, backspace, new-line, and bell are supported.
`void vb_getchar (c)` `int d;`	Get a typed character directly from the bios. You can not redirect input that is fetched using *vb_getchar( )*.

`void vb_puts(s, move)`
`char    *s;`
`int     move;`
	Write a string out to the screen. Move the cursor only if *move* is true.

`void vb_replace (c)` `int     c;`	Write a character to the screen without moving the cursor.
`int vb_inchar (attrib)` `int     *attrib;`	Return the character at the current cursor position. Modify **attrib* to hold the attribute byte associated with that character.
`int vb_getcur( )`	Return an integer representing the current cursor position. The top byte holds the row; the bottom holds the column.
`void vb_setcur(posn)` `int     posn;`	Send the cursor to a position fetched with a previous *vb_getcur( )* call.
`void vb_ctoyx (y,x)` `int     y, x;`	Set cursor position to *(y,x)* (row *y*, column *x*). The upper-left corner of the screen is *(0,0)*.
`void vb_getyx (&y, &x)` `int     *y, *x;`	Put the current cursor position into **y* (the row) and **x* (the column).
`int vb_iscolor( )`	Return true if the color monitor (as compared to the monochrome monitor) is installed.
`void vb_cursize (top,bot)` `int     top, bot;`	Set cursor size to extend from the top scan line to the bottom scan line of an individual character.
`void vb_blockcur( )`	Make the cursor a block cursor.
`void vb_normalcur( )`	Make the cursor a normal (underline) cursor.
`void vb_scroll(left,right,` `top,bot,amt)`	Scroll a region of the screen. *Left* is the column number of the leftmost column in the region, *right* of the rightmost; *top* is the top line, *bot* is the the bottom, and *amt* is the number of lines to scroll. If *amt* is positive, the region scrolls up, otherwise it scrolls down.

Table 1: *Functions in vbios.c*

(continued from page 95)

characters. Always call *noecho( )* at the top of your program and echo characters yourself. Another echo-related problem is caused by DOS. In order to get buffered input, you have to use a DOS function that always echoes. So, if *nocrmode( )* is active, the echo status is ignored.

The final two configuration subroutines are:

```
int nl ( );
int nonl ( );
```

When *nl( )* is active, a newline ('\n') is converted to a carriage-return, line-feed sequence on output, and a carriage return ('\r') is mapped to a newline on input; otherwise, no mapping is done. It's usually convenient to set *nl( )* at the top of your program.

Initializing Windows

Five functions are supported for initializing windows. You don't have to use any of them if your screen is one big window that occupies the whole screen. The first function is:

```
WINDOW *newwin( lines, cols, begin_y, begin_x )

int cols;
int lines;
int begin_y;
int begin_x;
```

which creates a new window *lines* rows high and *cols* columns wide with the upper-left corner at (*begin_y, begin_x*). [All coordinates here are (y,x), where *y* is the row number and *x* is the column number. The upper-left corner of the screen is (0,0).] The window is both created and cleared. A pointer to a *WINDOW* structure, declared in curses.h, is returned in a manner analogous to *fopen( )*. You must save this pointer to pass to other curses functions.

A variant on the *newwin( )* subroutine is:

```
WINDOW *subwin( win, lines, cols, begin_y, begin_x )
```

Here, *win* is a pointer to a window created with a previous *newwin( )* or *subwin( )* command. My implementation of curses treats the *subwin( )* command just like it does *newwin( )*. The real curses package, however, creates a subwindow. When a parent window is refreshed by curses, all subwindows are refreshed too. By the same token, if you read characters from a parent window, you'll be able to get characters from the subwindow as well. Similarly, when you delete a parent window, all the subwindows are deleted too.

The curses package supports a special *stdscr* window that represents the entire screen. This superwindow is created for you automatically by *initscr( )*. It's convenient to declare all other windows as subwindows to *stdscr* so that you can use the global functions discussed later. Note, however, that you may not pass *stdscr* as a *WINDOW* pointer to any of the other subroutines that take *WINDOW* pointers as arguments. The real curses package lets you do this, but mine doesn't support the practice. In fact, because no error checking is done in this situation, passing *stdscr* to a function results in a "Null pointer assignment" error message when your program terminates.

Three other subroutines affect an entire window. The macro:

```
scrollok (win, flag)

WINDOW *win;
int flag;
```

is passed a *WINDOW* pointer and a flag. If the flag is true, the indicated window is allowed to scroll; otherwise, the window does not scroll and characters that go off the bottom of the window are discarded. Note that line wrap (when you go off the right side of the window, you end up on the left edge of the next line) is always enabled. Scrolling is always enabled on the *stdscr* window.

The macros:

```
refresh( )
wrefresh(win)

WINDOW *win;
```

are used by the real curses to signal a screen refresh. They force the screen to coincide with the internal representations of the screen. No characters are actually written out to the terminal until a refresh occurs. My own curses package writes to the screen immediately, so both of these macros expand to null strings; in other words they are ignored. You'll need them to be able to port code to Unix, however. *Refresh( )* refreshes the whole screen (the *stdscr* window and all subwindows of *stdscr*), *wrefresh(win)* is passed a *WINDOW* pointer and refreshes only the indicated window. Note that the *refresh( )* command only works under the real curses if all windows are subwindows of *stdscr*.

The function:

```
int     box( win, vert ,horiz )

WINDOW *win;
int vert, horiz;
```

draws a box in the outermost characters of the window using *vert* for the vertical characters and *horiz* for the horizontal ones. I've extended this function to support the IBM box-drawing characters. That is, if IBM box-drawing characters are specified for *vert* and *horiz*, *box( )* uses the correct box-drawing characters for the corners. The box-drawing characters are defined in box.h as:

HORIZ (0xc4)	single horizontal line
D_HORIZ (0xcd)	double horizontal line
VERT (0xb3)	single vertical line
D_VERT (0xba)	double vertical line

Boxes can have double horizontal lines and single vertical ones, or vice versa. The Unix *box( )* function uses the vertical character for the corners.

Note that *box( )* doesn't draw a box around the window, as the Unix documentation would have you believe; rather, it draws the box in the outermost characters of the window itself. This means you can overwrite the border if your output lines are too wide. When you scroll the window, the box scrolls too. A function that creates a bordered window in which the border is not part of the window itself is shown on lines 592–613 of Listing Four, page 81. Here, two windows are created, one nested inside the other. The outer window just holds the box, and the inner window is used normally (for characters).

10 Important Reasons C Programmers Use Our File Manager

1. It's written in C.
Clearly the growing language of choice for applications that are fast, portable and efficient. All of db_VISTA's source code is written in C.

2. It's fast – almost 3 times faster than a leading competitor.
Fast access that comes from the unique combination of the B-tree indexing method and the "network" or direct "set" relationships between records. A winning combination for fast performance.

3. It's flexible.
Because of db_VISTA's combination of access methods, you can program to your application needs with ultimate design flexibility. Use db_VISTA as an ISAM file manager or to design database applications. You decide how to optimize run-time performance. No other tool gives you this flexibility without sacrificing performance.
db_VISTA is also well behaved to work with most any other C libraries!

4. It's portable.
db_VISTA operates on most popular computers and operating systems like UNIX, MS-DOS and VMS. You can write applications for micros, minis, or even mainframes.

5. Complete Source Code available.
We make our entire C Source Code available so you can optimize performance or port to new environments yourself.

6. It uses space efficiently.
db_VISTA lets you precisely define relationships to minimize redundant data. It is non-RAM resident; only those functions necessary for operation become part of the run-time program.

7. Royalty free run-time.
Whether you're developing applications for yourself or for thousands, you pay for db_VISTA or db_QUERY only once. If you currently pay royalties to someone else for your hard work, isn't it time you switched to royalty-free db_VISTA?

db_VISTA™

Features
- **Multi-user** support allows flexibility to run on local area networks
- **File structure** is based on the B-tree indexing method
- **Transaction processing** assures multi-user consistency
- **File locking** support provides read and write locks
- **SQL-based db_QUERY** is linkable
- **File transfer** utilities included for ASCII, dBASE optional
- **Royalty-free** run-time distribution
- **Source Code** available
- **Data Definition Language** for specifying the content and organization of your files
- **Interactive database access** utility
- **Database consistency check** utility

File Management Record and File Sizes
- Maximum record length limited only by accessible RAM
- Maximum records per file is 16,777,215
- Maximum file size limited only by available disk storage
- Maximum of 256 index and data files
- Key length maximum 246 bytes
- No limit on number of key fields per record
- No limit on maximum number of fields per record

Operating System & Compiler Support
- **Operating systems:** MS-DOS, PC-DOS, UNIX, XENIX, UNOS, ULTRIX, Microport, VMS
- **C compilers:** Lattice, Microsoft, IBM, DeSmet, Aztec, Computer Innovations, Turbo C, XENIX and UNIX

8. SQL-based db_QUERY™
Add our new C-linkable, SQL-based, ad hoc query and report-writing companion product to provide a simple relational view of your db_VISTA applications. Without compromising speed.

9. Free tech support.
60 days of free technical and application development support for every Raima product. Of course, extended support and training classes are also available at your place or ours.

10. Upward database compatibility
Start out with file management in a single-user PC environment—then move up to a multi-user LAN or a VAX database application with millions of records. You'll still be using db_VISTA. That's why so many C programmers are choosing db_VISTA.

But don't just take our word for it.
"Raima's customer support and documentation are excellent. Source code availability and royalty-free run-time is a big plus."
Dave Schmitt, President
Lattice, Inc.

"db_VISTA has proved to be an all-round high performer in terms of fast execution, flexibility and portability, and has undoubtedly saved us much time and development effort."
John Adelus, Hewlett-Packard
Office Productivity Division

30-day Money Back Guarantee!
Try db_VISTA in your environment for 30 days and prove it to yourself. If not completely satisfied, return it for a full refund.

Price Schedule

	db_VISTA	db_QUERY
☐ Single user	$ 195	$ 195
☐ Single user w/Source	$ 495	$ 495
☐ Multi-user	$ 495	$ 495
☐ Multi-user w/Source	$ 990	$ 990
NEW:		
☐ VAX Multi-user	$ 990	$ 990
☐ VAX Multi-user w/Source	$1980	$1980

Order Now.
Put db_VISTA to work in your application program. Ordering is easy—simply call toll-free. We'll answer your technical questions and get you started. Call today.

Call Toll-Free Today!
1 (800) db-RAIMA
(800/327-2462) or
206/828-4636

DD77

MasterCard VISA

RAIMA™ CORPORATION

3055 - 112th NE, Bellevue, WA 98004 USA
(206) 828-4636 Telex: 6503018237 MCI UW

UNIX TOOLS FOR YOUR PC

C CHEST
(continued from page 96)

This way, you can overflow the inner window and not affect the outer one (that holds the border).

Moving the Cursor

Three functions are supported for cursor movement:

```
int move ( y, x)
int wmove( win, y, x)
getyx ( win, y, x)

WINDOW *win;
int y, x;
```

Move( ) moves the cursor to the indicated absolute position on the screen. The upper-left corner of the screen is (0,0). *Wmove( )* moves the cursor to the relative position within a specific window (pointed to by *win*). The upper-left corner of the window is (0,0). If you try to move past the edge of the window, the cursor will be positioned on the edge. The *getyx( )* macro loads the current cursor position for a specific window into *y* and *x*. Note that this is a macro, not a subroutine, so you should not precede *y* or *x* with an address-of operator (&). This one command (only) accepts *stdscr* as a *win* argument. A *getyx(stdscr,y,x)* call loads the current absolute cursor position into *y* and *x*.

Keyboard Input

Two keyboard-input functions are supported:

```
int getch( )
int wgetch(win)

WINDOW *win;
```

Getch( ) just gets a character from the keyboard, and *wgetch( )* echoes the character to the indicated window (if *echo( )* is enabled, that is). Note that *crmode( )* has to be enabled to get the character as soon as it's typed, otherwise, the entire line will be buffered. It's unfortunate that many compiler manufactures (Microsoft included) have chosen to use *getch( )* as the name of their standard direct keyboard-input function, but so it goes.

Reading Back from the Screen

Three functions are supported for

reading back characters that are already on the screen:

```
inch( )
mvinch(y,x)
mvwinch(win,y,x)

WINDOW *win;
int y,x;
```

Inch( ) returns the character at the current cursor position. *Mvinch(y,x)* moves the cursor to the indicated position and then returns the character at that position; *mvwinch(win,y,x)* does the same but the cursor position is relative to the specified window. Some older versions of curses don't support the *mv* versions of this command.

Formatting Output
Two formatted output functions are supported:

```
printw (fmt, args . . . )
int wprintw (win, fmt, args . . . )

WINDOW *win;
char *fmt;
```

Printw( ) works just like *printf( )* does; *wprintw( )* is the same but it prints to the indicated window, moving the cursor to the correct position in the new window if necessary. (That is, it's moved to the position immediately following the character most recently written to the indicated window). *Printw( )* ignores window boundaries, but *wprintw( )* wraps when you get to the right edge of the window and the window scrolls when you go past the bottom line (provided that *scrollok( )* has been called for the current window).

Single Characters, Writing Strings
The single-character and string-write functions are:

```
addch(c)

int waddch(win, c)
int waddstr(win, str)

WINDOW *win;
int c;
char *str;
```

Addch( ) works like *putchar( )* does; *waddch( )* writes a character to the indicated window (and advances the cursor); and *waddstr( )* works like *fputs( )*, writing a string out to the indicated window. *Waddstr( )* does not add a *'\n'* at the end of the string. *Waddch( )* treats several characters specially:

'\n'—clear the line from the current cursor position to the right edge of the window. If *nl( )* is active, you go to the left edge of the next line; otherwise, you go to the current column on the next line. In addition, if scrolling is enabled, the window scrolls if you're on the bottom line.
'\t'—expand to an eight-space field. If the tab goes past the right edge of the window, the cursor wraps to the next line.
'\r'—move to the left edge of the window, on the current line.
'\b'—back up one space but not past the left edge of the window. Nondestructive. The curses documentation doesn't say that *\b* is handled explicitly, but it does indeed work.

The escape character is not handled specially by Unix, but my *waddch( )* does do so. (Don't use explicit escape sequences if portability is a consideration.) In particular, all characters between an ASCII ESC and an alphabetic character (inclusive) are sent to the output but are otherwise ignored. This lets you send escape sequences directly to the terminal if you like. I'm assuming here that you won't change windows in the middle of an escape sequence.

Erasing
Five erase functions are available:

```
werase(win)
erase( )

wclear( )
clear( )

wclrtoeol (win)
WINDOW *win;
```

Clear( ) and *erase( )* both clear the entire screen, *wclear( )* and *werase( )* both clear only the indicated window, and *wclrtoeol( )* clears the line from the current cursor position in the indicated window to the right edge of the indicated window.

Scrolling
Finally, two scrolling functions are supported:

```
scroll (win)
wscroll (win, amt)

WINDOW *win;
int amt;
```

Scroll( ) scrolls the indicated window up one line, and *wscroll( )* scrolls by the indicated amount—up if *amt* is positive, down if it's negative. This last function is not supported by the Unix curses.

There's one caveat about scrolling. The Unix functions have a bug in them in that, when a window scrolls, the bottom line is not cleared, leaving a mess on the screen. This problem is not restricted to the *scroll( )* subroutine but occurs any time that the window scrolls (as when you send a *\n* at the bottom line of the window or when a character wraps, causing a scroll. As a consequence, if you're porting to Unix, you should always do a *wclrtoeol( )* immediately after either scrolling or printing a new line. Unfortunately, there's no easy way to tell if a window has scrolled because of a character wrap. My curses package doesn't have this problem—the bottom line of the window is always cleared on a scroll.

Implementation
For the most part, the code is straightforward and needs little comment. The *WINDOW* structure is declared on lines 7–17 of curses.h (Listing Three). The macros for *bool*, *reg*, *TRUE*, *FALSE*, *ERR*, and *OK* are defined in the Unix curses.h file, so I've put them here too. Be careful of:

```
if( foo( ) == TRUE )
```

TRUE is *#define*d as 1, but in fact any nonzero value is true. As a consequence, *foo( )* could return a perfectly legitimate true value that didn't happen to be 1, and the test would fail. The test:

```
if( foo( ) != FALSE )
```

is safe, however. Most of the output functions return *ERR* if scrolling is disabled and the write would have caused a scroll.

MICRO/ SYSTEMS JOURNAL

FOR THE ADVANCED COMPUTER USER

SUBSCRIBE NOW AND

SAVE OVER 15%

OFF THE NEWSSTAND PRICE!

$5 SAVINGS

Yes! I want to subscribe to

Micro/Systems Journal™

and save over 15% off the cover price.

☐ 1 Year (6 issues) $20 ☐ 2 Years (12 issues) $35

☐ Please charge my: ☐ Visa ☐ MasterCard ☐ American Express

☐ Payment enclosed ☐ Bill me later

Card # ____________________ Exp. date __________

Signature ____________________

Name ____________________

Address ____________________

City ____________________ State ________ Zip __________

Savings based on a full one-year cover price of $23.70. Canada and Mexico add $3 for surface mail, $7 for airmail per year. All countries add $12 for airmail per year. All foreign subscriptions must be prepaid in U.S. dollars drawn on a U.S. bank. Please allow 6-8 weeks for delivery of first issue.

A Publication of M & T Publishing, Inc. 3014

$5 SAVINGS

Yes! I want to subscribe to

Micro/Systems Journal™

and save over 15% off the cover price.

☐ 1 Year (6 issues) $20 ☐ 2 Years (12 issues) $35

☐ Please charge my: ☐ Visa ☐ MasterCard ☐ American Express

☐ Payment enclosed ☐ Bill me later

Card # ____________________ Exp. date __________

Signature ____________________

Name ____________________

Address ____________________

City ____________________ State ________ Zip __________

Savings based on a full one-year cover price of $23.70. Canada and Mexico add $3 for surface mail, $7 for airmail per year. All countries add $12 for airmail per year. All foreign subscriptions must be prepaid in U.S. dollars drawn on a U.S. bank. Please allow 6-8 weeks for delivery of first issue.

A Publication of M & T Publishing, Inc. 3014

$5 SAVINGS

Yes! I want to subscribe to

Micro/Systems Journal™

and save over 15% off the cover price.

☐ 1 Year (6 issues) $20 ☐ 2 Years (12 issues) $35

☐ Please charge my: ☐ Visa ☐ MasterCard ☐ American Express

☐ Payment enclosed ☐ Bill me later

Card # ____________________ Exp. date __________

Signature ____________________

Name ____________________

Address ____________________

City ____________________ State ________ Zip __________

Savings based on a full one-year cover price of $23.70. Canada and Mexico add $3 for surface mail, $7 for airmail per year. All countries add $12 for airmail per year. All foreign subscriptions must be prepaid in U.S. dollars drawn on a U.S. bank. Please allow 6-8 weeks for delivery of first issue.

A Publication of M & T Publishing, Inc. 3014

No Postage
Necessary
If Mailed
In The
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT 790, REDWOOD CITY, CA

Postage Will Be Paid By Addressee

For the Advanced Computer User
Micro/Systems Journal

No Postage
Necessary
If Mailed
In The
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT 790, REDWOOD CITY, CA

Postage Will Be Paid By Addressee

For the Advanced Computer User
Micro/Systems Journal
Box 3713
Escondidio, CA 92025-9843

C CHEST
(continued from page 99)

The *mvinch( )* and *mvwinch( )* macros on lines 40 and 41 use the comma operator, often called the sequence operator. The comma operator evaluates from left to right, and the entire expression evaluates to the rightmost object in the list. For example, *mvinch( )* looks like:

```
#define mvinch(y,x)\
(move(y,x), inch( ))
```

An equivalent subroutine is:

```
mvinch(y,x)
{
   move(y,x);
   return inch( );
}
```

The comma operator is used because two statements have to be executed—the *move( )* call and the *inch( )* call. Were you to define the macro as:

```
#define mvinch(y,x) move(y,x); inch( )
```

the following code wouldn't work:

```
if( condition )
   mvinch(y,x);
```

because it would expand to:

```
if( condition )
   move(y,x);
inch( );
```

Putting curly braces around the statements doesn't help. For example:

```
#define mvinch(y,x)\
                        {move(y,x); inch( );}

if( condition )
   mvinch(y, x);
else
   something( );
```

expands to:

```
if( condition )
{
   move(y,x);
   inch( );
}
;
else
   something( );
```

Here the *else* will try to bind with the semicolon, which is a perfectly legitimate statement in C, causing a "No matching if for else" error message. Though the comma operator solves both of these problems, it isn't very readable. I don't recommend using it unless you must. Never use it if curly braces will work in a particular application.

The next problem is the *wprintw( )* function. (*Printw( )* is just a macro that evaluates to *printf( )*, so it isn't a problem.) In order to keep control of the window, you can't just blast characters to standard output; rather, characters must be sent through *waddch( )* so that you can position the cursor correctly, scroll the window when necessary, and so forth. The problem is solved by using a special formatting output function, called *vfprintf( )* in the ANSI standard and *_doprnt( )* by Unix. They both take three arguments:

```
_doprnt ( fmt, args, stream )
vfprintf( stream, fmt, args )
```

Fmt is the format string, *stream* is the output stream, and *args* is the address of the first argument in the argument list. For example, *fprintf( )* looks like this:

```
fprintf(stream, fmt, args)
FILE *stream;
char *fmt;
char *args;
{
   _doprnt( fmt, &args, stream );
}
```

Note that I've cheated here and not followed either the official ANSI or Unix methods of passing arguments to a subroutine with a variable number of arguments. The above example is easier to figure out, however. I've done it correctly in the code (on lines 345–361 of Listing Four). The process is also discussed in depth in my book *The C Companion* (see the bibliography) in which a complete source for *_doprnt( )* is presented. In fact, I've used the version from this book in curses (on line 359). This version is nonstandard in that it is passed a pointer to an output func-

(continued from page 101)

tion rather than a stream pointer.

It is possible to use the standard *_doprnt( )* or *vprintf( )* functions, even though both of these are passed pointers to output streams rather than pointers to output functions. The problem is solved by the versions of *doprnt( )* shown in Examples 1 and 2, right.

The first of these is ANSI compatible. It solves the output-function problem by formatting into a string—using the ANSI *vsprintf( )* function—and then writing the string out, one character at a time, using the function pointer that was passed as the first argument.

The solution for Unix systems (Example 2) is harder because there's no Unix equivalent to *vsprintf( )*. Here you have to send the formatted output to a file, rewind the file, and then read the file back, one character at a time, calling your output function to do the actual printing. The temporary file is created in the *if* statement at the top of the subroutine; *tmp_file* will be *NULL* the first time that *doprnt( )* is called. The temporary file name is created with a call to the Unix (and ANSI) *mktemp ("yyXXXXXX")* function, which creates a unique file name. *Mktemp( )* is passed a template for the name, and it replaces the last six characters in that template with a number guaranteed to form a unique name (one that isn't currently in use). If you aren't using a Unix- or ANSI-compatible compiler, just replace the *mktemp( )* call with some reasonable string (such as *"$$$$$$$$.tmp"*). Having created a target file, *doprnt( )* calls *_doprnt( )* to write out to that file. Finally, it writes out a *'\0'* to mark the end of string, rewinds the file, and then reads it back, printing each character. I'm assuming that your program will call *exit( )* to close the temporary file.

Note that, even though this code looks pretty awful, it's not as inefficient as it seems. Because I'm using the buffered read and write functions and because most strings are shorter than a buffer, there's virtually no disk activity. That is, all your reads and writes are really going to the internal disk buffer, not to the disk itself. Only those strings that are longer than the buffer should cause a disk read or write.

Erratum

Jack Whitney of Walnut Creek, California found an error in Table 1 of the February C Chest (page 96). The *if* statement in the HASHPJW function should look like this:

```
if( g = h & ~((unsigned) (~0) >> 4 ))
```

Nonetheless, the contents of the table are correct. The somewhat convoluted code is discussed in this month's Flotsam and Jetsam.

Bibliography

Arnold, Kenneth. "Screen Updating and Cursor Movement Optimization: A Library Package." *Unix Programmer's Manual,* vol. 2. This article is the documentation for the real curses package.

```
#include <stdarg.h>

static  doprnt(ofunct, funct_arg, fmt, argp)
int     (*ofunct)();
char    *funct_arg;
char    *fmt;
va_list *argp;
{
    /* A doprnt() for ANSI                              */
    /*          (c) Copyright 1987, Allen I. Holub.     */

    char        buf[133], *p ;

    vsprintf( buf, fmt, argp );
    for( *p = buf; *p; (*ofunct)( *p++, funct_arg ) )
        ;
}
```

Example 1: A doprnt( ) *for ANSI*

```
#include <varargs.h>

static  doprnt(ofunct, funct_arg, fmt, argp)
int     (*ofunct)();
char    *funct_arg;
char    *fmt;
va_list argp;
{
    /* A doprnt() for Unix.                             */
    /*          (c) Copyright 1987, Allen I. Holub.     */

    int         c                 ;
    extern char *mktemp()         ;
    static char *tmp_name         ;
    static FILE *tmp_file = NULL  ;

    if( !tmp_file )
    {
        tmp_name = mktemp("yyXXXXXX");

        if( !(tmp_file = fopen(tmp_name , "w+")) )
        {
            fprintf(stderr,"Can't open temporary file %s\n",
                                                tmp_name );
            exit( 1 );
        }
    }

    _doprnt( fmt, argp, tmp_file );

    putc    ( 0, tmp_file );
    rewind  (    tmp_file );

    while( (c = getc(tmp_file)) != EOF && c )
            (*ofunct)( c, funct_arg );

    rewind( tmp_file );         /* Get ready for next call */
}
```

Example 2: A doprnt( ) *for Unix*

C CODE FOR THE PC

source code, of course

C Source Code

FSP (screen manager)	$400
GraphiC 3.0 (high-resolution, DISSPLA-style scientific plots in color & hardcopy)	$300
Essential C Utility Library (400 useful C functions)	$160
Essential Communications Library (C functions for RS-232-based communication systems)	$160
Panache C Program Generator (screen-based database management programs)	$150
B-Tree Library & ISAM Driver (file system utilities by Softfocus)	$100
The Profiler (program execution profile tool)	$100
QC88 C compiler (ASM output, small model, no longs, floats or bit fields, 80+ function library)	$90
CBTree (B+tree ISAM driver, multiple variable-length keys)	$80
ME (programmer's editor with C-like macro language by Magma Software)	$75
Wendin PCNX Operating System Shell	$75
Wendin PCVMS Operating System Shell	$75
Wendin Operating System Construction Kit	$75
Entelekon C Function Library (screen, graphics, keyboard, string, printer, etc.)	$70
Entelekon Power Windows (menus, overlays, messages, alarms, file handling, etc.)	$70
EZ_ASM (assembly language macros bridging C and MASM)	$60
Make (macros, all languages, built-in rules)	$50
Vector-to-Raster Conversion (stroke letters & Tektronix 4010 codes to bitmaps)	$50
Coder's Prolog (inference engine for use with C programs)	$45
PC/MPX (light-weight process manager; includes preemption and cooroutine packages)	$45
Biggerstaff's System Tools (multi-tasking window manager kit)	$40
TELE Kernel (Ken Berry's multi-tasking kernel)	$30
TELE Windows (Ken Berry's window package)	$30
Clisp (Lisp interpreter with extensive internals documentation)	$30
Translate Rules to C (YACC-like function generator for rule-based systems)	$30
6-Pack of Editors (six public domain editors for use, study & hacking)	$30
ICON (string and list processing language, Version 6 and update)	$25
LEX (lexical analyzer generator)	$25
Bison & PREP (YACC workalike parser generator & attribute grammar preprocessor)	$25
C Compiler Torture Test (checks a C compiler against K & R)	$20
A68 (68000 cross-assembler)	$20
Small-C (C subset compiler for 8080 and 8088)	$20
tiny-c (C subsubset interpreter including the tiny-c shell)	$20
Xlisp 1.5a (Lisp interpreter including tiny-Prolog in Lisp)	$20
List-Pac (C functions for lists, stacks, and queues)	$20
XLT Macro Processor (general purpose text translator)	$15
C Tools (exception macros, wc, pp, roff, grep, printf, hash, declare, banner, Pascal-to-C)	$15

Data

Webster's Second Dictionary (234,932 words)	$60
U. S. Atlas (29,000 cities with retrieval program)	$40
KST Fonts (13,200 characters in 139 mixed fonts: specify TEX or bitmap format)	$30
The World Digitized (100,000 longitude/latitude points)	$30
NBS Hershey Fonts (1,377 stroke characters in 14 fonts)	$15
U. S. Map (15,701 points)	$15

The Austin Code Works
11100 Leafwood Lane
Austin, Texas 78750-3409
(512) 258-0785

Free shipping on prepaid orders CIRCLE 250 ON READER SERVICE CARD **MasterCard/VISA**

C CHEST
(continued from page 102)

Duncan, Ray. *Advanced MS-DOS*. Redmond, Wash.: Microsoft Press, 1986.
Holub, Allen. *On Command: Writing a Unix-like Shell for MS-DOS*. Redwood City, Calif.: M&T Publishing, 1987.
Holub, Allen. *The C Companion*. Englewood-Cliffs, N.J.: Prentice-Hall, 1987. *Printf( )* is discussed in depth in Chapter 9.
Norton, Peter. *The Peter Norton Programmer's Guide to the IBM PC*. Redmond, Wash.: Microsoft Press, 1985.

Availability

The code presented this month is destined for a book I'm writing (on the subject of compiler design). It is copyrighted by myself. Though you're welcome to both download it and use it yourself, you may not distribute it in any form (including binary) to anyone else or use it for any commercial purposes.

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to Dr. Dobb's Journal, 501 Galveston Dr., Redwood City, CA 94063, or call (415) 366-3600, ext. 216. Please specify issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 74.)

Flotsam and Jetsam

Portable Masks

Bit masks are used to set or clear individual bits in a number. For example *n & 0x1* clears all but the bottom bit of *n*. By the same token, *n | 0x1* sets the bottom bit (to 1). It's easy for bit masks to be nonportable, however. For example, if you want to clear only the bottom bit of a number, it's tempting to say: *n & 0xfffe* This statement makes an important assumption, however. It assumes that an *int* is 16 bits wide. If you tried to use it on a machine that had a 32-bit *int*, the top 16 bits of the number would be cleared too. You'd have to say *0xfffffffe* on a 32-bit machine.

You can correct this problem by using the one's complement operator (~), which reverses the sense of all bits in a word (maps the 1s to 0s and vice versa). For example, the expression *n & ~1* clears only the bottom bit of a word, regardless of the word width. On a 16-bit machine, *~ 1* evaluates to *0xfffe;* on a 32-bit machine, it evaluates to *0xfffffffe.*

A similar problem arises when you want to set or clear only the top bit of a word. For example, *n & 0x8000* sets the top bit of *n*; but it also only works with a 16-bit int. You'd have to say *n & 0x80000000* on a 32-bit machine. This problem can be solved with a little convoluted coding. The expression

```
~((unsigned) (~0) >> 1)
```

evaluates to 0x8000 on a 16-bit machine and 0x80000000 on a 32-bit machine. The *(~0)* evaluates to an int-size number in which all the bits are set. The cast to unsigned tells the compiler not to sign extend the number on the right shift. The shift then moves all the bits one notch to the right, shifting in a zero from the left (because it's unsigned). Finally, the outermost ~ reverses the sense of the mask.

Note that the cast is required here because many compilers treat *~0* as a signed integer (having the value -1). These compilers process the shift in one of two ways—both incorrect. If the compiler looks at *>> 1* as a divide by 2, *~0>>1* will evaluate to 0 (-1/2 should be 0). If the compiler treats the *>> 1* as an arithmetic right shift, it's likely to duplicate the top bit rather than shifting in a *0* (so *~0>>1* will do nothing).

The processed can be generalized. The *top_n_bits(n)* macro given below evaluates to a constant with the top n bits set:

```
#define top_n_bits(n)\
        ~((unsigned) (~0) >> n)
```

Note that all this shifting and inverting will be done at compile time by most compilers. That is, the expression will actually evaluate to a single constant in the generated code, not to a series of shift and invert instructions. Consequently, it's no less efficient than the more straightforward-looking, but less portable, variant.

Malloc()

Enough people have written about the March Flotsam and Jetsam, in which I discussed problems with the *malloc( )* and *calloc( )* system calls, so that a little more discussion of the problem seems in order. First, *malloc( )* is unique in that it is guaranteed to return a pointer that can point at any sort of object. You can't always assume that pointers are the same size, regardless of the object to which they point. For example, in the 8086 medium model, you can have a 16-bit pointer to a subroutine and an 32-bit pointer to data (or vice versa). For the same reason, it's a mistake to cast a pointer in to an *int* because, if your machine has a 32-bit pointer and a 16-bit *int*, the value will be truncated. In some machines you can't even assume that pointers to two data objects will be the same. The problem here is alignment. A compiler for a machine that requires *long*s to be aligned on 4-byte boundaries—and *int*s on 2-byte boundaries—is likely to round a pointer to *long* when it is cast into a pointer to *int* (in order to maintain alignment). In general, it's not portable to cast a pointer into anything. Of course, it's not always possible to avoid this sort of cast. *Printf( )*, for example, often has to make certain assumptions about pointers and these assumptions are often nonportable. The foregoing notwithstanding, you can always cast the return value of *malloc( )* into any kind of pointer—but only because *malloc( )* was written with this sort of portability in mind. By the same token, an *extern* statement can always be used to declare *malloc( )* or *calloc( )* as returning a pointer to any type of object. Don't do this with other subroutines, however—at least not if you want your code to be portable.

A TOOLBOX OF C-SOURCE CODE THAT LETS YOU GET YOUR "EGA" APPLICATION UP AND RUNNING QUICK!

EGA GRAPHICS TOOLS

INCLUDES:

- *Lines, Circles, Circle Segments, Arcs, Squares and Polygons*
- *Save/Load Screens and Windows to/from Disk*
- *Window Management, Moving Windows*
- *Color and Palette Changes*
- *Draw Axis, Plot Points*
- *Rotates and Moves*
- *Multiple Fonts*
- *Plus More...*

ONLY $59.95

Cobra Systems

14700 Main Street, Suite 3, Bellevue, Wa 98007 (206)641-2759

Add $5.00 shipping & handling. Washington residents add 8.1% sales tax.

CIRCLE 245 ON READER SERVICE CARD

UNIX-LIKE TOOLS

Lock into the POWER of UNIX with CCtools, THE Programmer's Toolkit™

CCtools contains various UNIX-look-a-like utilities and text processing commands useful in any programmer's toolbox. Some of these programmer productivity tools are fgrep, head, tail, qpr, cat, wc, more, od, getopt, line, od, page, path, touch, cp, uniq, true, false, expr, tee, and others.

All programs within **CCtools** come with complete documentation and can be run from the normal system prompt. For instance, via command.com on an MSDOS machine. **CCtools** is a must for the serious programmer.

CCtools is currently available on various machines for UNIX, MSDOS, and other non-UNIX environments at the introductory price of only $24.95! We can be reached at 718-849-2355 if you have any questions.

We offer a FREE hotline, and a 30-day unconditional refund policy. We will never sell a copy-protected disk. No shipping, handling or hidden costs.

Comeau Computing
91-34 120th Street
Richmond Hill, NY 11418

Member of the Programmer's Co-op

CIRCLE 211 ON READER SERVICE CARD

FORTRAN PROGRAMMERS

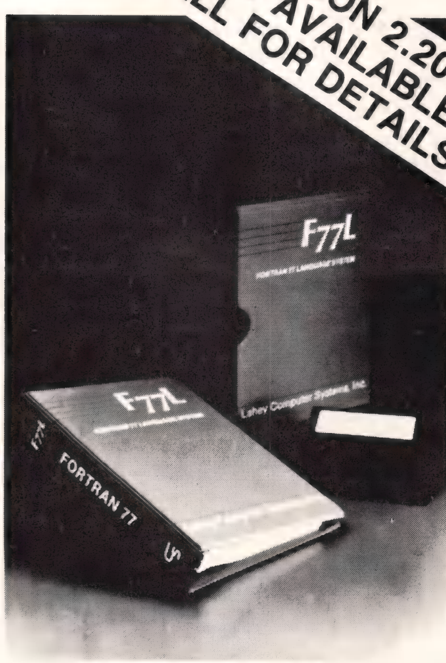
VERSION 2.20 NOW AVAILABLE CALL FOR DETAILS

Looking for the right PC FORTRAN language system? If you're serious about your FORTRAN programming then you should be using F77L - LAHEY FORTRAN.

"Lahey's F77L FORTRAN is the compiler of choice. It's definitely a 'Programmers FORTRAN,' with features to aid both the casual and the professional programmer. . . F77L compiled the five files in a total of 12 minutes, which was 4 times as fast as MS FORTRAN and an astounding 6 times as fast as Pro FORTRAN." - PC Magazine

Compare the features and performance of other PC FORTRANs with F77L and you will find that F77L is clearly the superior product.

- Full Fortran 77 Standard (F77L is not a subset)
- Popular Extensions for easy porting of mini and mainframe applications
- COMPLEX*16, LOGICAL*1 and INTEGER*2
- Recursion - allocates local variables on the stack
- IEEE - Standard Floating Point
- Long variable names - 31 characters
- IMPLICIT NONE
- Fast Compile - Increases productivity
- Source On Line Debugger (Advanced features without recompiling)
- Arrays and Commons greater than 64K
- Clear and Precise English Diagnostics
- Compatibility with Popular 3rd Party Software (i.e. Lattice C)
- Easy to use manual
- Technical Support from LCS
- *NEW FEATURE - NAMELIST*



Editor's Choice
- PC Magazine

F77L - THE PROGRAMMER'S FORTRAN

$477.00 U.S.

System Requirements: MS-DOS or PC-DOS, 256K, math coprocessor (8087/80287)

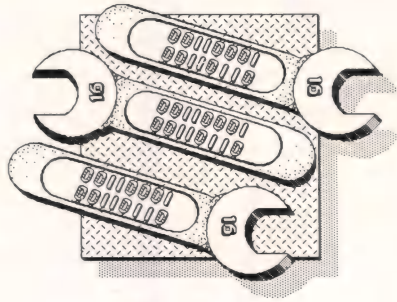
FOR MORE INFORMATION: (702) 831-2500

Lahey Computer Systems, Inc.
P.O.Box 6091
Incline Village, NV 89450
U.S.A.

International Dealers:
England: Grey Matter Ltd., Tel: (0364) 53499
Denmark: Ravenholm Computing, Tel: (02) 887249
Australia: Computer Transitions, Tel: (03) 537-2786
Japan: Microsoftware, Inc., Tel: (03) 813-8222

SERVING THE FORTRAN COMMUNITY SINCE 1967

CIRCLE 186 ON READER SERVICE CARD



80386 Programming Tools

by Ray Duncan

Perhaps Santa's reindeer are getting old, or perhaps all the personal computers hanging off the back of his sleigh create too much drag. At any rate, he didn't arrive with my Compaq 386 Deskpro until the end of January—but the suspense was worth it. Remember when you switched from a 4.77-MHz IBM PC to a PC/AT or compatible? For the first few days, I was continually reminded of what a two to three times speed increase can do for one's productivity as a software developer. Fortunately, the human organism is highly adaptable, and I have already learned to take the speed of the 386 for granted.

Microsoft has warned the computing industry not to expect an 80386-specific version of protected mode DOS until at least late 1988. Programmers, like other children of nature, abhor a vacuum, and several companies are attempting to exploit this window of opportunity and carve out a niche for themselves in the 80386 marketplace. MetaWare is marketing C and Pascal compilers, Quarterdeck Systems is shipping an 80386-specific version of the DESQview control program, several versions of 80386 Unix/Xenix are underway, and The Software Link (a company previously known for its copy-protection schemes) is about to release a multitasking operating system called PC/MOS 386.

The first real 80386 programming tool to fall into my clutches (it actually arrived before the Compaq did) was the Phar Lap 80386 assembly-language development package 386ASM / 386LINK, which includes the following programs:

- 386ASM.EXE: a full-fledged macro assembler supporting all 8086, 80186, 80286, 80386, 8087, 80287, and 80387 opcodes.
- 386LINK.EXE: a linker that produces a load module in the "old". EXE format.
- MINIBUG.EXE: a program debugger capable of running 80386 real mode or protected mode applications, with a command set equivalent to MS-DOS DEBUG (except for the *A* command).
- RUN386.EXE: a protected mode runtime environment for 32-bit 80386 applications running under MS-DOS; Phar Lap calls this program the 386 DOS-Extender.

The distribution disks also contain various example programs and some object modules that allow the Phar Lap linker to be used with the Microsoft C compiler. The 375-page manual is clearly written with lots of examples. Like the Microsoft MASM manual, it concerns itself primarily with the assembler pseudo-ops or directives; readers are referred to the Intel *80386 Programmer's Reference* for information about the CPU instruction set and the syntax for the instruction mnemonics.

Writing a new protected mode 32-bit 80386 application with these tools, or converting an existing application to take advantage of 32-bit protected mode, is quite simple. The attribute *USE32* must be included in each of the segment declarations in the module. This tells the assembler that the 32-bit override prefix byte is not required before instructions that perform 32-bit operations. After assembly, the program is linked with a special module (START386.OBJ), supplied by Phar Lap, that flags the application as capable of being run in 32-bit protected mode and prevents it from being accidentally run in real mode (if the resulting .EXE file is run directly from the MS-DOS command line, it simply displays an error message and exits).

The MINIBUG and RUN386 tools are then used to debug or run the 32-bit protected mode application under MS-DOS. The applications can request MS-DOS system services in the normal way—that is, by loading the 80386 registers with appropriate values and performing an *Int 21h*. The debugger or RUN386 intercepts the software interrupt, performs any address translation that may be required, switches the CPU into real mode, and transfers to MS-DOS to carry out the desired function. Upon return from MS-DOS, the debugger or RUN386 performs any necessary translation on returned values, switches the CPU back into protected mode, and gives control back to the application. MINIBUG and RUN386 also supply the application with segment selectors that allow it to access command tail arguments, physical memory from 0 to 640K, and the video refresh buffer directly.

Phar Lap will soon be releasing a "binding" for 32-bit protected mode applications that performs the same function as the RUN386 program but that can be linked right into the application—making the presence of the RUN386.EXE file unnecessary. The price of this binding will be $995, and it will include the right to distribute the Phar Lap code as an integral part of an application without further royalties or license fees.

By now, you are undoubtedly wondering about the Phar Lap tools' compatibility and performance compared to the Microsoft Macro Assembler (MASM) and Linker (LINK.EXE). On paper, 386ASM is nearly 100 percent upward compatible with the Microsoft Macro Assembler; the only ex-

ceptions being lack of support for the *.ERR1*, *.ERR2*, and *.CREF* directives. The few new 80386-specific directives—*.386C*, *.386P*, *.387*, *DP* (a 6-byte data type for 32-bit protected mode address pointers)—and the segment attributes *USE16* and *USE32* are stylistically consistent with the Microsoft MASM syntax.

In practice, the upward compatibility of 386ASM from Microsoft MASM for 8086/286 applications indeed proved to be excellent. I assembled and linked several large source files for my own company's MS-DOS-based products, and the only differences I detected lay in the area of more stringent error checking on the part of 386ASM. For example, 386ASM reported the source code line:

```
esc equ 01bh
```

as an error because of the collision between the symbol *esc* and the Intel instruction mnemonic *ESC*. Microsoft MASM let the same line pass without comment and interpreted the equate "as expected" when it was referenced later in a line such as:

```
cls db esc,'[2J'
```

The Phar Lap assembler has two particularly nice features that are not supported by the Microsoft assembler. The first is the ability to redirect error messages to a file or device distinct from the file or device that receives the entire program listing. The other is support for local labels; the scope of any label starting with the # character is limited to the current procedure.

To test 386LINK's compatibility with Microsoft LINK, I used it to rebuild a relatively large assembly-language application containing some 250 object modules from existing libraries that had been created with Microsoft's LIB, MASM, and C and LMI's UR/FORTH object module compiler. The resulting .EXE file was not identical on a byte for byte basis with the .EXE file produced by Microsoft LINK, presumably because of slightly different library search or segment building strategies in the two linkers, but it was exactly the same size (19,414 bytes) and ran correctly.

It should be noted that in order to support the 80386's 32-bit (native) instructions, Phar Lap has extended the standard Intel Object Module Format (OMF-86) with new classes of 32-bit offsets, displacements, and fixups. These extensions are similar but not identical to the OMF extensions made by Microsoft in its XENIX/386 Toolkit.

Command-line compatibility between the Phar Lap tools and the Microsoft tools is something else again. Phar Lap opted to design a completely new command-line syntax for 386ASM and 386LINK and did not make any attempt to follow the Microsoft conventions. For example, the MASM command:

```
C>MASM FILE1,FILE2;
```

which assembles FILE1.ASM to create FILE2.OBJ and does not create a listing or cross-reference file, would look like this:

```
C>386ASM FILE1 -OBJECT FILE2 -NOLIST -8086
```

for the Phar Lap assembler (unlike MASM, it creates a listing file by default). Similarly, the Microsoft LINK command line:

```
C>LINK /NOI MAIN+MENU,MYFILE,,NUCLEUS
```

which links the modules MAIN.OBJ and MENU.OBJ with the modules in the library NUCLEUS.LIB and produces the executable module MYFILE.EXE, would be entered as:

```
C>386LINK MAIN MENU -EXE MYFILE -LIB NUCLEUS -8086 -TWOCASE
```

for the Phar Lap linker. I feel that this lack of command-line compatibility was not a sound strategic decision by Phar Lap. Aside from the fact that the Microsoft command syntax is considerably terser and thus more efficient, everyone is already familiar with it. Adoption of the same syntax would have made transition to the Phar Lap tools that much easier, and existing MAKE files could be used without modification. Perhaps if enough people complain, Phar Lap will cave in and make the change before it builds up too big a user base.

The Phar Lap assembler has two particularly nice features not supported by the Microsoft assembler.

Performance of the Phar Lap tools is acceptable, but they are not nearly as lean and mean as their Microsoft equivalents. 386ASM.EXE weighs in at 215K, compared to 85K for MASM 4.0, and 386LINK is 75K, as opposed to 48K for Microsoft LINK. As might be expected for such plump programs, they are also slow. The following timings were obtained on a Compaq 386 Deskpro (16-MHz, 2-megabyte RAM, 70-megabyte fixed disk):

Assembly of a 600-line assembler source file with a few simple macros and three segments:
- Microsoft MASM, Version 4.0: 6.2 seconds
- Phar Lap 386ASM, Version 1.1d: 10.8 seconds

Linkage of a 19K .EXE file containing approximately 250 object modules:
- Microsoft LINK, Version 3.51: 18.6 seconds
- Phar Lap 386LINK, Version 1.1c: 2 minutes 14.4 seconds

A spokesman for Phar Lap says that the company's own timings on its linker indicate it is only about 30 percent slower than the Microsoft linker, so the application I was linking may represent some sort of worst case. Still, I suspect that the performance of 386LINK, together with the incompatible command syntax, will deter people from converting all their work onto the Phar Lap tools—including the development of 8086 real mode applications—a conversion that would be perfectly feasible given the otherwise high degree of compatibility between Phar Lap and Microsoft's products.

For the first release of such sophisticated programming tools, the Phar Lap products are remarkably sound, and we can reasonably expect that

(continued from page 107)

they will continue to evolve and improve. Phar Lap's technical support is above reproach, too. I had several occasions to send E-mail questions to Phar Lap via its account on BIX, and in each case I received helpful replies within the same day.

In any event, 386ASM and 386LINK are currently the only game in town if you want to start working with the 80386 in its native 32-bit mode and don't hanker to throw away half your CPU cycles and hard disk on Xenix. The price of the MS-DOS version of the Phar Lap assembler/linker/debugger package is $495. Versions that run under VAX VMS or Unix and cross-assemble and link to the 386 are also available. Phar Lap can be contacted at 60 Aberdeen Ave., Cambridge, MA 02138; (617) 661-1510.

Book Corner

Prompted by visionaries such as Jean Yates, who prophesied that Unix would be running on everything but your toaster oven by 1990, publishers have been flooding the bookstores with "advanced" Unix books for the last two years. Most of these books pan out to be collections of Unix E-mail tricks, warnings not to leave your terminal unattended while logged in, or guides to bigger and better shell scripts (batch files to you MS-DOS types). A relatively few books deliver something more substantial and address the Unix application program interface.

Books that actually describe the internals of Unix in any detail, especially those that do it in a way comprehensible to a normal mortal, have been virtually nonexistent, however. The only readily available resource of any worth, aside from the source code for Unix itself (if you have $50,000 to spare for it), has been the two special issues of the *AT&T Bell Labs Technical Journal* (July/August 1978 and October 1984), which were devoted to Unix articles by various Unix program authors, pioneers, gurus, and mystics.

This deficiency has been decisively remedied with the appearance of *The Design of the UNIX Operating System*, by Maurice J. Bach.[1] Mr. Bach works at Bell Labs and based his book on Unix System V, Release 2, source code, though some coverage is given to BSD Unix variants as well. He covers the kernel architecture, file system, control of processes, interprocess communication, device drivers, and even multiprocessor Unix systems. The book is thorough and well written, but it will be heavy going for readers lacking previous exposure to Unix and a general understanding of operating system and hardware concepts such as processes, scheduling, kernel and user mode, interrupt handlers, memory protection, swapping, page faults, and so on.

An even more welcome book is *Operating Systems: Design and Implementation*, by Andrew S. Tanenbaum,[2] which easily qualifies as the best general book on operating systems I have ever seen. It covers all the necessary subjects (processes, file systems, interprocess communication, memory management, mass storage, and so on) in the context of a Unix-like operating system for the IBM PC called MINIX.

MINIX has the same system calls as Version 7 Unix, as well as a shell and some 60 other system utilities compatible at the user level with Unix, but the source code is completely original and is included in the book (also available on diskette). An interesting feature of MINIX is that the file system manager runs outside the operating system as a user process. Consequently, it can be easily modified and even replaced with a file server process that accesses un-Unix-ish file structures (MS-DOS perhaps) or performs reads and writes across a communications link or network.

Unlike many Unix authors, Tanenbaum has broad experience with other operating systems and frequently draws examples and comparisons from his knowledge of VM/370, MULTICS, and so on. The discussions of processes, scheduling, interprocess communications, and deadlocks are particularly coherent. I recommend this book without reservation to anyone interested in the principles of operation of modern operating systems.

MASM Equates

In the February 1987 column, I pointed out some subtle differences be-

SAS Institute Inc. Announces

Lattice C Compilers for Your IBM Mainframe

Two years ago...
SAS Institute launched an effort to develop a subset of the SAS® Software System for the IBM Personal Computer. After careful study, we agreed that C was the programming language of choice. And that the Lattice® C compiler offered the quality, speed, and efficiency we needed.

One year ago...
Development had progressed so well that we expanded our efforts to include the entire SAS System on a PC, written in C. And to insure that the language, syntax, and commands would be identical across all operating systems, we decided that all future versions of the SAS System—regardless of hardware—would be derived from the same source code written in C. That meant that we needed a C compiler for IBM 370 mainframes. And it had to be good, since all our software products would depend on it.

So we approached Lattice, Inc. and asked if we could implement a version of the Lattice C compiler for IBM mainframes. With Lattice, Inc.'s agreement, development began and progressed rapidly.

Today...
Our efforts are complete—we have a first-rate IBM 370 C compiler. And we are pleased to offer this development tool to you. Now you can write in a single language that is source code compatible with your IBM mainframe and your IBM PC. We have faithfully implemented not only the language, but also the supporting library and environment.

Features of the Lattice C compiler for the 370 include:

- **Generation of reentrant object code.** Reentrancy allows many users to share the same code. Reentrancy is not an easy feature to achieve on the 370, especially if you use non-constant external variables, but we did it.
- **Optimization of the generated code.** We know the 370 instruction set and the various 370 operating environments. We have over 100 staff years of assembler language systems experience on our development team.
- **Generated code executable in both 24-bit and 31-bit addressing modes.** You can run compiled programs above the 16 megabyte line in MVS/XA.
- **Generated code identical for OS and CMS operating systems.** You can move modules between MVS and CMS without even recompiling.
- **Complete libraries.** We have implemented all the library routines described by Kernighan and Ritchie (the informal C standard), and all the library routines supported by Lattice (except operating system dependent routines), plus extensions for dealing with 370 operating environments directly. Especially significant is our byte-addressable Unix®-style I/O access method.
- **Built-in functions.** Many of the traditional string handling functions are available as built-in functions, generating in-line machine code rather than function calls. Your call to move a string can result in just one MVC instruction rather than a function call and a loop.

In addition to mainframe software development, you can also use our new cross-compiler to develop PC software on your IBM mainframe. With our cross-compiler, you can compile Lattice C programs on your mainframe and generate object code ready to download to your PC.

With the cross-compiler, we also offer PLINK86™ and PLIB86™ by Phoenix Software Associates Ltd. The Phoenix link-editor and library management facility can bind several compiled programs on the mainframe and download immediately executable modules to your PC.

Tomorrow...
We believe that the C language offers the SAS System the path to true portability and maintainability. And we believe that other companies will make similar strategic decisions about C. Already, C is taught in most college computer science curriculums, and is replacing older languages in many. And almost every computer introduced to the market now has a C compiler.

C, the language of choice...
C supports structured programming with superior control features for conditionals, iteration, and case selection. C is good for data structures, with its elegant implementation of structures and pointers. C is conducive to portable coding. It is simple to adjust for the size differences of data elements on different machines.

Continuous support...
At SAS Institute, we support all our products. You license them annually; we support them continuously. You get updates at no additional charge. We have a continuing commitment to make our compiler better and better. We have the ultimate incentive—all our software products depend on it.

For more information...
Complete and mail the coupon today. Because we've got the development tool for your tomorrow.

SAS®

SAS Institute Inc.
SAS Circle, Box 8000
Cary, NC 27511-8000
Telephone (919) 467-8000 x 7000

I want to learn more about:

☐ the C compiler for MVS software developers
☐ the C compiler for CMS software developers
☐ the cross-compiler with PLINK86 and PLIB86

today...so I'll be ready for tomorrow.

Please complete or attach your business card.

Name ______________________
Title ______________________
Company ______________________
Address ______________________
City ______________ State ________ ZIP ________
Telephone ______________________

Mail to: SAS Institute Inc., Attn: CC, SAS Circle, Box 8000, Cary, NC, USA. 27511-8000. Telephone (919) 467-8000, x 7000

DDJ 7/87

(continued from page 108)

tween *equ* and = that can lead to unexpected problems. Steve Russell of SLR Systems writes:

"There are lots of peculiarities between *equ* and =. The differences are barely hinted at in the manual. The example in the manual [*MASM 4.0 Reference Manual*, page 55] that says:

```
clearax equ xor ax,ax
```

doesn't work if you try to use it, though *clearax* does show up in the symbol table as being equal to the text value.

"The distinction between the *equ* and = pseudo-ops is a little more subtle than suggested in your column. For instance, if you remove the supposedly redundant *offset*s and just use:

```
dw $-test
```

you find that *equ* and = yield identical results (to each other, not to your example)—that is, *equ* stores the value, not the text. On the other hand, if you simply write:

```
t1 equ offset test
t2 = offset test
```

then try:

```
mov ax,t1
```

and:

```
mov ax,t2
```

you get two different results. It looks as though *offset* cannot be stored as part of a symbol, so *equ* stores text if the expression does not yield a 'value' and the = directive throws away *offset* because it can't store text.

"So the question is, is the latter action a bug in = because it faithfully throws *offset* away without saying so or is it an undocumented feature?"

The Dialogue Goes On

Frank Albe, of Houston, Texas, takes up the cudgel this month on the subject of high-level vs. assembly languages:

"Charles Lyall's letter in your March column has been nagging me for a week now, so I decided it's time to respond. As you pointed out, Mr. Lyall's letter is well crafted and states some informed opinions.

"The letter makes four major points:

"1. You can't master as many assembly languages as you can high-level languages (HLLs).
"2. It takes orders of magnitude more time to write in assembly languages than it does in HLLs.
"3. The difference in execution time does not warrant investing the extra programmer time.
"4. Fourth-generation languages (4GLs) and HLLs will trample assembly languages in the dirt because the cost-accountant mentality says, 'Go for the lowest bid for equivalent function.'

"To me, this is the voice of an application analyst. From his vantage point, his arguments are very persuasive and full of common sense. My position is that of an opinionated systems software developer who 'hacked his first piece of code' in 1963 on a CDC 1604:

"1. It is a lot more difficult to be fluent in many assembly languages because we have to know more than the assembly language, per se. We must develop deeper understanding of the target operating system and hardware than is required of the person who writes exclusively in HLLs. In my opinion, HLL applications can benefit dramatically from careful attention to these details, but far too few HLL programmers are willing to invest the time to gain the knowledge and make good use of it.
"2. Programmers always underestimate the time it takes to write a functioning program and to burnish it to their satisfaction. I don't know anything about the example TEE, but unless it's pretty trivial, I doubt that most people could write it from scratch in 15 minutes in an HLL such as C, Pascal, COBOL (snicker, snicker), or FORTRAN. For the purposes of argument, assume the relationships of 8 to 1 for coding and 10 to 1 for execution time hold across the board. I know this is invalid, but these numbers are as good as any others to make my point. If the program will run infrequently and not in conjunction with others in an interactive suite, the 10-second load and execute time is probably acceptable. If the program is a tool such as the *DIR* command in MS-DOS, a perceivable delay is intolerable.
"3. The total elapsed time to get code operational is a significant factor. In the example, it is reasonable to assume that one person can accomplish either task in a single sitting at a terminal. It is a matter of taste and professional judgment which is better: blindingly fast execution or incredibly quick production of the source code.

"Let's jump from the sublime to the ridiculous and consider a complex cost-accounting application that must be run daily on existing hardware. It will take 2 years to develop in assembly language and will run 3 hours. If you need it in 3 months, can you accept a 30-hour daily run time? On the other hand, can you stay in business 2 years while the software engineers develop the Great Golden Wheel, and will you still need the application then? I know there are fallacies in this hypothesis, but it illustrates the point that it's not all black and white. There are many colors and shades of gray out there.

"We can't discount performance and compactness just because our personal computers are getting bigger and faster, lest we be doomed to relive the third-generation mainframe era. I hope there are enough responsible newcomers who will learn the history and folklore of this age of dinosaurs and profit from it.
"4. I always prefer to program in an HLL wherever practical, but it will take at least one more generation of hardware and/or software before we reach the point where assemblers are unnecessary. Reduced instruction set computers look like the best bet right now. We seem to be using the RISC models as an excuse to justify an evolution in compiler technology that reduces the compiler's scope and pushes the work down to a common object code optimizer. The theory, of course, is that you can write one of these to support all your compilers. This concept is not new, it's just getting a lot of good press right now.

"The most important benefit will be realized when the vast majority of executable code is truly reentrant. I have an inflatable soapbox that I've been carrying around since 1971, upon which I have preached this sermon many times.

"When my HLL code is properly optimized and reentrant, I will gladly clear storage and disavow all knowledge of any assembly language. Until that time, however, I reserve the right to write in an assembly language of my choice when I feel the results are justified. The same holds true for the HLL of my choice."

Erratum

In my May discussion of Command Plus from ESP Software Systems I gave an incorrect phone number for the company. The correct numbers are (213) 390-7408 (in California) and (800) 992-4377 (outside California).

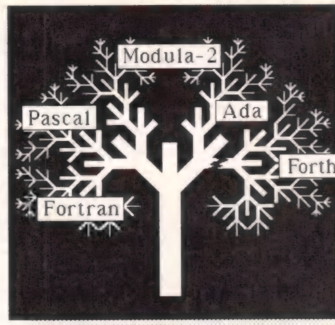
Notes

1. Maurice J. Bach; *The Design of the UNIX Operating System* Inc., Englewood Cliffs, N.J.: Prentice-Hall, 1986. ISBN 0-13-201799-7 025.
2. Andrew S. Tanenbaum, *Operating Systems: Design and Implementation*, Englewood Cliffs, N.J.: Prentice-Hall, 1987. ISBN 0-13-637406-9 025.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 6**.

Software Design Rules



Forth was forged for the purpose of programming: not to teach programming, not to serve as an environment for learning, not to match pre-existing notational conventions. Charles Moore and the Forth developers designed tools—and tools to create tools. Forth's ad hoc development, rooted in experimentation and use, resulted in a comfortable fit with both the structure of the computer and the process of problem solving.

I was reminded of this as I read *Programmers at Work*, a recent book by Susan Lammers (Redmond, Wash.: Microsoft Press, 1986). This collection of intriguing interviews with a selected group of innovative programmers reveals in their work some Forth-like tracks from time to time. I previously had encountered instances of similar convergent evolution, when programmers demonstrating a software development package with a surprisingly Forth-like structure stressed that they had not known or copied Forth but had arrived at the structure independently, finding it the optimal solution to their programming problem.

Given Forth's power as a general tool and its use-based development, I believe them. A solution found once is likely to arise again, especially if it is optimal: the various paths to a solution converge at the optimum.

An instance of convergence is

by Michael Ham

found in Bob Carr's comments on his work and design ideas in creating Framework. He talks, for instance, about granularity: "Users must be able to break their work, and the program, into separate pieces rather than dealing with a single, giant entity." Granularity and "homogeneity" (meaning that the architecture does not have a lot of exceptions or special cases) amount to what is called factoring in the Forth culture.

Factoring is not unique to Forth, of course. My speculation is that in Forth some tools are presented particularly clearly because of Forth's workaday evolution. Because these tools approach the optimal, they are likely to have been discovered in other contexts as well.

Carr mentions that he "had to steal a terrific design notion Xerox originated: All commands should act on data already selected or highlighted by the user. It is called the object-verb design, versus the verb-object design." Forth users will recognize that Forth uses the object-verb design extensively—putting the object on the stack and then executing the verb. But Forth did not take the idea from Xerox. The idea arises in the search for solutions, and experience directed different paths to that common optimum.

As Carr points out, the object-verb sequence shows its power by the degree to which it can do more work with fewer mechanisms. "Intuitive" often means "what I am used to," so I don't call the sequence intuitive. The sequence can be learned, of course, to the extent that it becomes second nature (witness the number of fans of Hewlett-Packard calculators), but the real argument in its favor comes from experience, which shows the power of the object-verb sequence and explains its emergence in various contexts other than Forth (and under various names).

Programmers at Work will be interesting reading for any programmer. Forth programmers will find not only various ideas familiar from Forth (though perhaps under different names) but also some mentions of Forth itself. Jef Raskin, for instance, discusses using Forth in his recent projects "because Forth is a rather compact language and is inexpensive to implement. It's not my favorite language, but I thought it was suitable for this particular application. I always believe you should use the right tool for the job."

Reading the book made me ruminate on the design rules that I follow. Here they are, hewn from my own experience.

The User Interface Is Everything

No matter how elegant the code, how ingenious the data structures, how efficient the file access, if the user interface is crude, clumsy, or confusing, the system will fail. The user interface refers to all interactions between users and the system: paper forms, computer messages, input procedures—all communication between users and the programs you have designed.

The system procedures must fit users' inclinations. Things must be done in a "natural" way, with no penalties for guessing or experimenting. How do you know what is natural for users? You can't ask them; you have to watch them. Watch them before you design the system, and watch them as they use your system. Pay careful attention to what causes them to stumble or hesitate. You must distinguish hesitations that stem from accidental awkwardness (for example, the arrangement of the furniture, which can be rearranged, or habits derived from the current procedure) and hesitations that stem from intrinsic awkwardness (for ex-

ample, an unreachable key combination or a complex instructional sequence for a common task).

The system must be comfortable for users for two reasons: first, to minimize error, and second, to encourage users to trust the system. If the system doesn't feel right and reliable, it will not be used.

This rule presents a particular challenge to programmers whose work habits were acquired writing batch programs in a mainframe environment. There, the users are computer professionals and semiprofessionals: operators, production clerks, and even other programmers. Although the requirement for a good user interface still holds, the sophistication and long-time experience of that user group can compensate to some degree for poor interface design. Moreover, the batch environment typically lacks interactive interfaces, in which users are confronted immediately and directly with the effects of poor design.

Users of microcomputer programs, on the other hand, are often not computer professionals. They are apt to be unfamiliar with the peculiarities of computer operation. Moreover, microcomputer programs are typically interactive in format and work with users face-to-face, as it were, instead of sending and receiving notes. These users depend on your design to make things go right for them.

Use Hindsight Early and Often

Design awhile, then stop a bit and review what you have done. With hindsight, you can see how you could have done something better. Revise, do more, and use hindsight again.

Note that, to use hindsight, you have to do something. It is good to plan and to know where you're going, but if you spend too much time in planning the details, you won't have time to redo parts of the system after experience shows where it needs revision and enhancement. Don't forget: no matter how well you plan, when you show the final product to the users, you will be told, "That is wonderful. But I guess I forgot to tell you this, and we just found out we're going to have to have that, and could you move that over, and I don't think we'll use this part after all."

To produce software that truly fits users' needs, you will learn to value iteration. You'll find that you must make closer and closer approximations to what your customers or clients want, or to what the situation requires, simply because the first fit (or the second) is never perfect.

Plan for iteration and use it. Don't hole up and work alone until you have the final, polished product and then show that to the users: it will not in fact be the final product, and you will have wasted time completing one developmental cycle when you could have completed several iterations that work toward what is really needed.

It's an excellent idea to write the manual before you start the system.

Design the Output First

Using an iterative approach does not mean that you start with no plan at all. You must have some idea of where you're going, and that means that you and the user should first agree on the destination (the output), even if you subsequently agree to change it. If you work through several drafts of the output, until both you and the user are satisfied with the format, content, definitions, sequence, and timing, you won't find yourself designing input forms or procedures that collect data you never use or (even worse) that fail to collect data that you need. Design proceeds backward through the system, implementation forward.

Document for You

If you have clear documentation, well organized and readable, it means you truly understand the system and, because it is understandable, it will be robust and easy to develop and maintain. The primary value of documentation is the process—working out in your own mind what you are doing and how you will do it. The secondary value is in having a record of what you have done, when you return to the system to make the inevitable changes.

And what about users' documentation? It is an excellent idea to write the users' manual before you start the system and let users read it to verify that the system operation makes sense in terms of the procedures and system objectives. Users will indeed read it: not yet having the system and curious about what you're planning, most users will pore over the document. But after the system is up, the users' documentation should (ideally) be unnecessary. Users will then generally go to the system, not to the manual. The acid test is whether the system itself (without documentation) can lead naive users correctly through standard operations while protecting itself against casual errors.

I am not suggesting that all application programs can meet this standard. Many programs have special features that make documentation necessary. But even those programs will fare better if their essential documentation is only a page.

If users say your documentation is hard to read and confusing, believe them. Don't go through the odd (but unfortunately common) exercise of trying to prove to them that the documentation is clear. If users cannot read or understand the documentation, the documentation is unacceptable. If users find the program hard to use, the program is indeed hard to use. If someone doesn't like liver, the statement "You just never had it cooked properly" seldom convinces them that liver is delicious.

Make the Program Bulletproof

Random pecking of keys should not cause a catastrophe. The input routines should filter out all dangerous or irrelevant input, and the program should also be alert for every internal awkwardness it might encounter in operational use: zero divisors, trying to append records from file A to file B when A = B, the wrong diskette being used as the input master file, trying to sort a single record, and

(continued from page 113)

so forth.

Avoid maddening tricks. For instance, don't ask users to confirm an action that subsequently proves impossible to do. Example: A user enters a request to delete record 357. The program responds "You wish to delete record 357 (Y/N)?" and the user dutifully responds "Y." The program then responds, "Record 357 not on file. Request aborted." The user rightfully feels tricked. If the record can't be deleted, the program should have said so in the first place instead of going through the miniquiz on intention. Even better, the confirmation question will display information from the record to the user, asking whether this is the record to delete. This approach satisfies both needs: the user is informed if there is no record to delete or, if the record is present, can readily confirm that it is the right record to delete.

Another example: In one program the user has the option of appending one file to another, and several things can go wrong with that request: the resulting file might be larger than legal for this particular application; the user may have specified the same file twice (absentmindedly trying to append a file to itself); or the two files may have had one or more elements in common, not allowed in this application. The program should check for all three possibilities before asking the user to confirm that file A is to be appended to file B; if the files cannot be appended, the appropriate error message should be displayed instead of the request for confirmation. This is only common sense, of course, but it is common sense applied from the user's viewpoint. The user's viewpoint must also be one of the designer's viewpoints.

I particularly dislike messages such as "Invalid option" or "Nonnumeric. Reenter." A good program should tactfully ignore inappropriate data. If only numeric input is valid, only numeric keys should function. Pressing any other key should produce no effect at all. The same goes for menu selection. "Invalid option" should never be necessary because the program should recognize only valid keys.

Users should be able to tell from the lack of action that something is wrong. If A, B, and C are the only valid data, then only the A, B, and C keys should work—and they should work equally well whether capitalized or not.

The program must meet users' reasonable expectations. For example, in one program it is necessary to determine the user's sex. For consistency with earlier menus, the user is asked to respond to the menu "1—Female, 2—Male," but the program accepts F, f, M, and m in addition to 1 and 2. Similarly, a menu consisting of "1—Yes, 2—No" should accept 1, 2, Y, y, N, or n—and even a carriage return if the default answer is clearly specified. An L typed when a numeral is expected is undoubtedly meant as a 1; why not accept it? Don't let the user complain, "It should have known what I meant." When the program should have known, make sure it does know.

The program thus hides within itself responses to all anticipated user inputs, including users' errors. The better job the designer does in anticipating users' moves, the more pleasant the program is to use. Users typically are not even aware that the program has responded from some option that lay in wait for the anticipatable error or intention. Success is achieved when the program responds as users expect, even when they do not do precisely what was asked. The options that users recognize will be only the tip of the iceberg.

Let Users Know What's Happening

Invalid input should evoke a response if it helps users. For example, if users attempt an illegal deletion of some sort, the program must so inform them, lest they get the impression that the deletion was accomplished. Or if the datum is a number that must lie within a range, the program should respond to an invalid entry by asking for reentry. The request for reentry should state the valid range because the users' invalid input suggests they may not know the valid range.

A nice example is the defining word *LIMITS* that a friend added to his Forth. *LIMITS* defines data input commands and expects at the time of definition the limits on the input range: *1 10 LIMITS PITCH* defines the command *PITCH* with limits *1* and *10*. When *PITCH* is executed, it displays "Enter PITCH" and waits for a number (using a word such as *DIGITS*, defined in the April 1987 Structured Programming column). The entered number is accepted only if it is within the limits. If it is not, an informative error message is displayed: "Input out of range: lower bound 1, upper bound 10. Reenter number." *LIMITS* is used to define any command expecting bounded numeric input.

Cute messages pall quickly. Be businesslike and brief, and always keep users informed. Long, silent waits—such as 5 seconds—make users uneasy. Tell them what's going on: "Program loading"; "Checking records"; whatever. When possible, display a countdown so users can estimate the rate of progress.

Be Consistent

No matter how much trouble it is, take pains to be consistent in every way possible: punctuation, significance of colors, mode of input, location of messages, and so on. If your menus are numbered lists from which users make a selection by entering the appropriate number, don't suddenly switch to a list in which they must enter the initial letter of the command. If the message "Press space bar to continue" appears in position 25 of line 20 in one screen, it should be in the same place—if possible—on every screen in which it is used. Users will quickly become accustomed to its location and expect to see it there. Users feel most secure when their expectations prove reliable.

Consistency doesn't come easily. For one thing, different sections of the program will have been written at different times. As a result, you must run through the "final" draft many times to be sure that it is consistent—that it feels the same in all the subroutines, that it embodies a consistent design and approach. Consistency comforts users and makes your product seem more trustworthy. It also makes the system easier to use and less likely to be a cause of error.

Give Every Routine a Safe Exit

Because users are (probably) using the program without reading the documentation (or with only a vague recollection of it), the program must have no traps: routines that, once entered, cannot be escaped from until something is done—for instance, adding a record or deleting one. If users select "Add record" from the menu, the add-record template should offer in the first entry a possible "escape" value that, when used, returns them to the menu. The Escape key can be used as a generic escape.

Suppose that the first entry in adding a record is the serial or ID number. A blank serial or ID number is an obvious way to escape the routine. The most general escape command is, of course, the "undo" key, which retracts the last command given. The undo may not be feasible in a given application, but some escape mechanism must be provided.

The escape value also makes it easy for users to work through batched input. Few users will add one record, delete a second, revise a third, add a fourth, run a list, back to another deletion, and so on. They will normally work through a group of new records, adding them all, then turn to a batch of revisions and, finally, deletions—working through all cases of a particular type. In such applications, the system should automatically return to do another action of the last type until users signify a desire to escape that routine.

One observation from experience: users usually consider deletion as a kind of revision, so your revision routine should usually include a way to delete the record instead of (or in addition to) a separate delete routine.

Watch Users Use Your System

I suggested this before, but it bears repeating. You cannot design a good system solely through your ideas of how things should be done. You also need to learn how they are done. You may learn that a particular application has users who do not batch input. By seeing how the system is used, you can polish it to remove any impediments it offers them, and you may also be able to suggest substantial changes that will make their work easier. Don't wait for users to think up improvements.

Obviously, if users do suggest an idea, listen closely. Users almost certainly understand the ins and outs of the job better than you do. But you can't shift the burden of good design onto their shoulders. Your job is to make the system responsive to their needs. This always requires close observation, which will sometimes lead not just to refinements but also to major changes.

Many users, for example, don't know what they really need; most will talk about the means rather than the goal. Because the designer is often thinking also about how to do things, it is easy to assume the goal and design the "how," rather than rigorously focusing first on "why." Systems that simply automate the existing clerical functions are an example of looking only at the how. Systems that radically redefine a process or final product, simplifying the entire procedure, are the result of repeatedly asking, "Why? Why do you need that? What do you do with it? What is its purpose? Who uses it? What do they do with it? What is it the means to?" and then finding the most efficient way to achieve the overall goals and objectives.

Design Programs from the Top Down, Experiment from the Bottom Up

I sometimes call this "sideways development" because it is not completely top-down or bottom-up. The top-down design goes well if you write a functional analysis of the entire program in a chunk of several words, then a functional analysis of those words, and so on:

```
PROGRAM: INTRO BEGIN INITIALIZE
                    MAIN  UNTIL DONE ;
INTRO: GET.DATE  CHECK.DATA.
                           DISKETTE ;
INITIALIZE:  FIRST.MENU  CASE NEW
           .TEST OLD.TEST MERGE.TEST
                 QUIT.WORK  ENDCASE ;
MAIN: BEGIN SECOND.MENU CASE AP-
       PEND REVISE REF.LIST SUMMARY
         QUIT.TEST  ENDCASE  UNTIL ;
```

and so forth.

If your programming language

STRUCTURED PROGRAMMING
(continued from page 115)

provides the tools, your design and exploratory implementation of the elements of the design should be concurrent: bottom-up coding to prototype and test solutions, which are then woven into the top-down design process.

The top-down descriptive analysis encourages you to see the program or system in terms of its major logical divisions, to see those divisions in terms of their major parts, and so on. This leads to a clean design that is easy to write, to read, and to maintain and that keeps you from getting lost, unable to see the forest for the trees. Writing the analysis will not be easy, and you will probably have to go through several drafts of each major function before they fit comfortably together.

The bottom-up exploratory programming will help you understand the problem better by allowing you to test tentative solutions. Exploratory programming works well only in truly interactive languages, in which the write-compile-test-revise cycle is uninterrupted by waits. In noninteractive languages, exploration is hindered by the overhead of preparing the source file, running the compiler to create an object file, linking the object file, and so on. These mechanical requirements disrupt your train of thought and discourage quick checking of short procedures. Exploration in such languages tends to become a premature production of long procedures rather than a quick interplay with small and simple prototypes.

Note that moving the colons in the phrases shown earlier to the start of each phrase produces Forth definitions. Forth was designed to lend itself to exploratory programming. Its structure and ability to accept new commands makes for an easy transition from the functional analysis to source code. The exploratory process produces a tested set of elemental commands and functions fitted to the problem, and the top-down analysis provides the high-level words for the final program. You then can start at the bottom level of the analysis and enter the elements defined in your exploration and the phrases defined in your analysis, testing your way back up the chain until you reach the final definition, which is the program.

See How the Flow of Activities Wants to Go

Look under the surface to find the natural sequence and direction of events. The actual procedure in place is only an approximation (and sometimes a poor approximation) of the "true" procedure, the ideal center that has pulled the actual procedure into its current configuration. The true procedure is the procedure in perfect focus; the actual procedure is always an approximation: slightly blurred, slightly off center. As the implemented procedure approaches the true procedure, things work more smoothly because effort is not spent in countering the natural tendencies of the work.

If users consistently make some error in the data-entry procedures, it is a sign that the procedures are wrong. If users must stop to calculate whether a set of data meets the requirements for the next program step, you should immediately suspect that the program itself should make the check.

You can readily find examples in noncomputer systems as well. The Army Corps of Engineers stopped the flow of sand down the Atlantic Coast, and then they found the southerly beaches vanishing as the sand washed away and was not replaced by sand from the north. Now bulldozers and dollars work to maintain beaches once renewed through natural processes. Because the system's natural flow was disrupted, much effort is devoted to a poor approximation of beach renewal.

Sometimes, to encourage a certain flow of events, you can design seeming inefficiencies into a procedure. For example, papercutting machines used in binderies can also cut people. One way to prevent accidents is to hire floor supervisors who constantly watch the operators and jerk them back if their hands stray too near the blade. A better way is to build the machines so that two switches, instead of one, must be closed to make the cut. Although it is easy and even cheaper to design the machine so that only a single switch is used, using two switches occupies both the operator's hands while the blade cuts. The second switch acts as an aileron in the flow of events, pulling the natural sequence in the direction the designer wants it to go.

Poorly designed systems, which diverge markedly from their natural center and course, exhibit great turbulence from the continual efforts required to keep the procedure on course. In a large system, the turbulence may be manifested as many supervisors or frequent meetings or reruns or down time or correction passes. Well-designed systems, on the other hand, will flow smoothly and swiftly and with almost no supervision or visible effort. The system itself pulls any inadvertent deviations back into the natural flow and thus is self-correcting. The feedback into well-designed systems keeps them on course; the feedback in poorly designed systems pushes them further off course, making users exercise constant vigilance and effort to keep the system running.

These rules are stated as if you were designing a system for a customer or client, but in fact the user will often be yourself. An iterative approach will almost always produce the most satisfactory system or program. You certainly want, for yourself, a system that doesn't require extensive documentation to run, a system that is robust and easy to maintain. Treat work for yourself with the same professional care that you would devote to work meant for others. Not only will you get better software but you will also acquire the habit of thoughtful and alert design.

Envoi

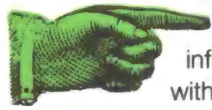
A new job and new responsibilities have forced me to choose among the activities I can do. I have enjoyed the opportunity to discuss Forth-related topics in these pages, and in the future I may have occasion to speak out again. But my contributions as a regular columnist end with this column. My current work includes an introductory book on programming in Forth, so my involvement with this fascinating language continues.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**

For Free Info ...

Start Here



Smart buyers start with *DDJ*'s free information card, a shopping center filled with information about the products and services advertised in this very issue: everything from software and systems to peripherals and professional support services.

And smart buyers can use this free information card to quickly and easily gather a comprehensive file of facts, figures and product specs to sort out competing claims. Using *DDJ*'s free information card can prevent you from making the wrong, costly buying decision.

Be a smart shopper. Complete and mail this postage paid card today!

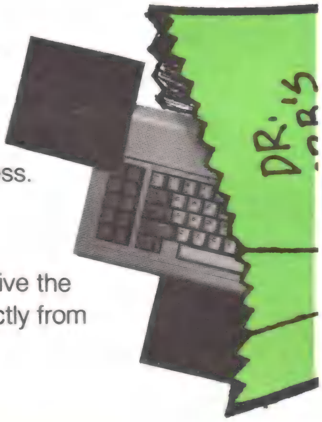
Take a Reader Service Card with You

It's Easy as ...

1. Circle the appropriate free information numbers, referring to the advertiser index for more information.

2. Fill in your name and address.

3. Mail today—postage is absolutely free. You'll receive the product information you need directly from the manufacturers, thanks to *DDJ*.



NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT #217, CLINTON, IOWA

POSTAGE WILL BE PAID BY ADDRESSEE

Dr. Dobb's Journal of Software Tools
FOR THE PROFESSIONAL PROGRAMMER

P.O. Box 2157
Clinton, Iowa 52735-2157

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT #217, CLINTON, IOWA

POSTAGE WILL BE PAID BY ADDRESSEE

Dr. Dobb's Journal of Software Tools
FOR THE PROFESSIONAL PROGRAMMER

P.O. Box 2157
Clinton, Iowa 52735-2157

Dr. Dobb's Journal of

Software Tools

FOR THE PROFESSIONAL PROGRAMMER

Use this card for FREE, FAST information about the products and services listed in this issue. Simply circle the appropriate numbers below.

1	26	51	76	101	126	151	176	201	226	251	276	301	326	351	376
2	27	52	77	102	127	152	177	202	227	252	277	302	327	352	377
3	28	53	78	103	128	153	178	203	228	253	278	303	328	353	378
4	29	54	79	104	129	154	179	204	229	254	279	304	329	354	379
5	30	55	80	105	130	155	180	205	230	255	280	305	330	355	380
6	31	56	81	106	131	156	181	206	231	256	281	306	331	356	381
7	32	57	82	107	132	157	182	207	232	257	282	307	332	357	382
8	33	58	83	108	133	158	183	208	233	258	283	308	333	358	383
9	34	59	84	109	134	159	184	209	234	259	284	309	334	359	384
10	35	60	85	110	135	160	185	210	235	260	285	310	335	360	385
11	36	61	86	111	136	161	186	211	236	261	286	311	336	361	386
12	37	62	87	112	137	162	187	212	237	262	287	312	337	362	387
13	38	63	88	113	138	163	188	213	238	263	288	313	338	363	388
14	39	64	89	114	139	164	189	214	239	264	289	314	339	364	389
15	40	65	90	115	140	165	190	215	240	265	290	315	340	365	390
16	41	66	91	116	141	166	191	216	241	266	291	316	341	366	391
17	42	67	92	117	142	167	192	217	242	267	292	317	342	367	392
18	43	68	93	118	143	168	193	218	243	268	293	318	343	368	393
19	44	69	94	119	144	169	194	219	244	269	294	319	344	369	394
20	45	70	95	120	145	170	195	220	245	270	295	320	345	370	395
21	46	71	96	121	146	171	196	221	246	271	296	321	346	371	396
22	47	72	97	122	147	172	197	222	247	272	297	322	347	372	397
23	48	73	98	123	148	173	198	223	248	273	298	323	348	373	398
24	49	74	99	124	149	174	199	224	249	274	299	324	349	374	399
25	50	75	100	125	150	175	200	225	250	275	300	325	350	375	400

July '87: Use before October 31, 1987

Name ______________________

Title ______________________

Company ______________ Phone ________

Address ______________________

City/State/Zip ______________________

Dr. Dobb's Journal of

Software Tools

FOR THE PROFESSIONAL PROGRAMMER

Use this card for FREE, FAST information about the products and services listed in this issue. Simply circle the appropriate numbers below.

1	26	51	76	101	126	151	176	201	226	251	276	301	326	351	376
2	27	52	77	102	127	152	177	202	227	252	277	302	327	352	377
3	28	53	78	103	128	153	178	203	228	253	278	303	328	353	378
4	29	54	79	104	129	154	179	204	229	254	279	304	329	354	379
5	30	55	80	105	130	155	180	205	230	255	280	305	330	355	380
6	31	56	81	106	131	156	181	206	231	256	281	306	331	356	381
7	32	57	82	107	132	157	182	207	232	257	282	307	332	357	382
8	33	58	83	108	133	158	183	208	233	258	283	308	333	358	383
9	34	59	84	109	134	159	184	209	234	259	284	309	334	359	384
10	35	60	85	110	135	160	185	210	235	260	285	310	335	360	385
11	36	61	86	111	136	161	186	211	236	261	286	311	336	361	386
12	37	62	87	112	137	162	187	212	237	262	287	312	337	362	387
13	38	63	88	113	138	163	188	213	238	263	288	313	338	363	388
14	39	64	89	114	139	164	189	214	239	264	289	314	339	364	389
15	40	65	90	115	140	165	190	215	240	265	290	315	340	365	390
16	41	66	91	116	141	166	191	216	241	266	291	316	341	366	391
17	42	67	92	117	142	167	192	217	242	267	292	317	342	367	392
18	43	68	93	118	143	168	193	218	243	268	293	318	343	368	393
19	44	69	94	119	144	169	194	219	244	269	294	319	344	369	394
20	45	70	95	120	145	170	195	220	245	270	295	320	345	370	395
21	46	71	96	121	146	171	196	221	246	271	296	321	346	371	396
22	47	72	97	122	147	172	197	222	247	272	297	322	347	372	397
23	48	73	98	123	148	173	198	223	248	273	298	323	348	373	398
24	49	74	99	124	149	174	199	224	249	274	299	324	349	374	399
25	50	75	100	125	150	175	200	225	250	275	300	325	350	375	400

July '87: Use before October 31, 1987

Name ______________________

Title ______________________

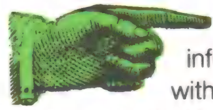
Company ______________ Phone ________

Address ______________________

City/State/Zip ______________________

For Free Info ...

Start Here



Smart buyers start with *DDJ*'s free information card, a shopping center filled with information about the products and services advertised in this very issue: everything from software and systems to peripherals and professional support services.

And smart buyers can use this free information card to quickly and easily gather a comprehensive file of facts, figures and product specs to sort out competing claims. Using *DDJ*'s free information card can prevent you from making the wrong, costly buying decision.

Be a smart shopper. Complete and mail this postage paid card today!

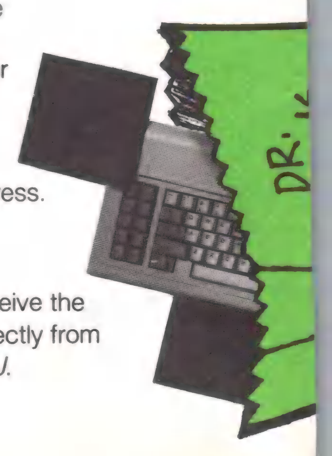
Take a Reader Service Card with You

It's Easy as ...

1. Circle the appropriate free information numbers, referring to the advertiser index for more information.

2. Fill in your name and address.

3. Mail today—postage is absolutely free. You'll receive the product information you need directly from the manufacturers, thanks to *DDJ*.



The Advertiser Index

*This advertiser prefers to be contacted directly; see ad for phone number.

Advertising Sales Offices

Southeast/Midwest
Gary George (404) 378-1396

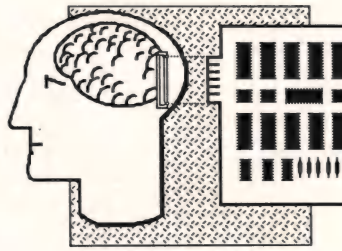
Northern California/Northwest
Lisa Boudreau (415) 366-3600

Northeast
Cynthia Zuck (718) 499-9333
Martha Brandt (415) 366-3600

Southern California/AZ/NM/TX
Michael Wiener (415) 366-3600

Director of Marketing and Advertising
Ferris Ferdon (415) 366-3600

The Xerox 1186 LISP Machine



This month I'll describe the Xerox 1186, an AI workstation, or LISP machine, I've been working with recently. In the process I hope to show why and how a piece of dedicated hardware is optimized for a language such as LISP and share some programming insights that have come out of LISP machine development.

The Xerox 1186, nicknamed Daybreak, provides several unique, powerful features at a relatively low cost. It is the result of several years of experience Xerox has had in producing advanced workstations intended specifically for interactive, exploratory programming in LISP for AI applications and research.

But LISP machines didn't start at Xerox.

How LISP Machines Came to Be

LISP machines grew out of the hacker culture that prevailed at the MIT AI lab in the 60s. You'll find the story of those days told in accessible terms in Steven Levy's book *Hackers* (Dell 1984). Richard Greenblatt, one of the main "hacker heroes" in Levy's book, invented the LISP machine. You'll find the technical history in his article, "The LISP Machine," in the book *Interactive Programming Environments* (Barstow, David, et al.,

by Ernest R. Tello

McGraw-Hill 1984).

But why design a special computer just to run the LISP programming language? The motivation was partly technical, partly pure hacker culture. The main technical problem that Greenblatt and his colleagues were trying to solve was that, because most serious AI programs tend to be rather large, they need a particularly efficient environment for execution and development. The other consideration that strongly influenced the LISP machine developers was the hacker antipathy to time sharing. The LISP machine had to be a personal computer rather than a time-shared machine so that the LISP hackers could have full control of the machine's resources. They had experienced what this meant with the older PDP-1 and PDP-6 machines, and they had seen and been impressed by the Alto personal computer that had been built at Xerox PARC, the first microcoded personal computer and the precursor of both the current generation of LISP machines and personal computers alike.

One way in which the machine could be optimized for LISP was by providing a large virtual memory architecture. Because of the large size of many AI programs, it was (at the time) a necessity for such programs to reside in virtual memory. Because the performance of large disk drives used on large time-shared machines had little better performance than those that could be used on small, personal machines, Greenblatt argued that, for virtual memory systems, personal, single-user machines were more appropriate and cost-effective.

The most peculiar aspect of LISP implementation to emerge from the development of LISP machines was the CDR-coding scheme used for list storage. The linked lists that are the heart of LISP code and data storage typically take up twice the space that comparable arrays require. The strategy of CDR coding is based on the fact that many of the lists in a LISP program are never or seldom modified. In this case, why waste the memory for the extra cell? The storage system Greenblatt arrived at gets the best of both worlds. As long as a list is not modified, it is stored as an array. Once the list is modified, fast microcode routines reassign the section of the list, from the point of the modification on, into the normal linked-list cell format in high memory.

CDR Coding, Tagged Architectures, and Invisible Pointers

There are at least five major requirements for efficient symbolic processing systems—that is, for an efficient LISP system—compact storage of linked-list structures; fast function-calling mechanisms; rapid run-time type checking; fast, incremental garbage collection; and efficient and powerful virtual memory management. The current generation of LISP machines is based on an architecture that addresses all these issues with an ingenious and elegant strategy. The basic ingredients of this strategy are a tagged architecture, CDR coding, invisible pointers, custom processors designed to enable microcoding of high-level instructions, and incremental garbage collection algorithms.

The original CONS and CADR machines developed at MIT used a 32-bit word size to implement the tag field CDR-coding strategy. As illustrated in Figure 1, page 121, the compressed list storage format was based on dividing the 32-bit data word into four different segments: a 24-bit data area for representing the first element of a list or CAR, a 5-bit data type tag field, a 2-bit CDR-code tag field, and a 1-bit garbage collection (GC) tag. The 2-bit CDR-code tags are used to signify the four values *CDR-NEXT*, *CDR-NIL*, *CDR-*

(continued from page 118)

NORMAL, and *CDR-ERROR*. This scheme allows list structures to be stored as ordinary vectors with a space savings of close to 50 percent. When a list is stored as an ordinary array, each word in the array carries the *CDR-NEXT* tag, except for the last item in the list, which is tagged *CDR-NIL*. When the list must be modified and when the sequential storage can no longer be maintained, the tag of the last word prior to the point of modification is changed to *CDR-NORMAL*, which indicates that the normal list storage format is now being used. In this case, the next word forms the second cell in the standard CONS cell pointer structure.

With this storage scheme, however, there is still a major drawback. When list modifications are made, the accessing functions can be slow and inefficient because many of the list elements must be moved into a fresh area of memory. This problem was solved by the invention of the invisible pointer. An invisible pointer is an indirect addressing scheme that is implemented on the level of the data itself rather than through an instruction. It's called invisible because there is no way for most of the system to see that the indirection is occurring. Only the lowest-level, memory-referencing operations handle the invisible pointers.

The invisible pointers make "closure" possible—the linking of internal value cells to external value cells. No macroinstructions are required to do this. When the system seeks to read or write to an element that has been restored in the CONS cell format, it is automatically sent by the invisible pointer to the new location. In this way the linkage of elements is maintained as efficiently as possible with no need to be concerned with it on the programming level.

Incremental garbage collection means taking out a little bit of garbage frequently so that you never reach the point at which things have to completely shut down while a large pileup of garbage is taken out. Most LISP systems today claim to have this capability.

The Xerox 1186

The 1186 closely resembles an earlier machine from Xerox—the 1108, or Dandelion. It runs all the same software, but the hardware has been streamlined using up-to-the-minute VLSI technology to provide a surprising amount of power for such a small package. The processing unit of the 1186 is a compact, vertical standing module 21½ inches high, 9½ inches wide, and 12½ inches deep. The large 19-inch, high-resolution monochrome display monitor is about the size of four Macintosh screens, offering a resolution of 862×1152 pixels. A 15-inch monitor is also available with a 632×833-pixel resolution. Both of these monitors have the same pixel density, though—80 pixels per inch. Color bit-map routines are supplied with the software for using graphics with a color display. The keyboard on the 1186 is the same as that used on the Xerox word-processing workstation and can be readily assigned all the keys of the PC series computers.

The main CPU of the 1186 uses a TTL bit-slice processor based on a high-speed version of the Advanced Micro Devices 2901C chip, which has been augmented with custom LSI and

gate arrays for microinstruction latching and bus decoding and arbitration. It also uses a front-end processor, based on the Intel 80186, and an option is available that uses a second 80186 to permit IBM PC emulation. The 1186 comes with a minimum of 1.6 megabytes of RAM, which is expandable to 3.5 megabytes, and hard disks are available in 20-, 40-, and 80-megabyte configurations. A variety of bus interface options are available that allow use of IBM PC, Multibus, and IEEE-488 peripherals. The power requirement for the 1186 is about 800 watts.

As with all LISP machines, the 1186 employs a virtual memory system whose behavior is largely responsible for the performance quality users experience. Virtual memory is provided in pages of 512K each and is allocated in two-page chunks called quanta. A total of 4 to 5 megabytes of virtual memory is required for the system to "say hello," and typical applications need between 8 to 10 megabytes. There is no such thing as enough RAM with this kind of machine; the only way to get maximum performance is with maximum RAM.

With a 16-bit address bus, the 1186 cannot use the sort of tagged architecture that I described earlier and that is found on many LISP machines. The virtual memory architecture is built on an ingenious 32-bit addressing scheme, however. The 1186 reads in the 32 bits in two chunks, 16 bits at a time, but the CPU does not have to wait for the second 16 bits to know which address is referenced.

The 1186 uses a form of incremental garbage collection, which is currently the feature that everyone claims to have.

The Programming Environment

On the 1186, the main interpreter, or LISP listener, is called the Executive and is accessed in its own window. One of the special features of the interactive LISP Executive is the Programmer's Assistant.

Pressing the right mouse button in a desktop bit-map area always results in the main pop-up menu opening, which allows you to access a variety of different facilities, depending on the options installed. On the machine I evaluated, which had the LOOPS AI system installed,

Data	Type	CDR Code	GC Tag
0 23	24 28	29 30	31

Figure 1: *Original MIT implementation of tag field CDR coding in a 32-bit word*

(continued from page 121)

the menu looked like that shown in Example 1, below. Menu items with an arrow head to the right sprout submenus off to the right if you drag the mouse across them. Windows on the 1186 are completely independent of one another and can be written to, in principle, by any number of processes.

Any window or icon on the 1186 also has an associated standard menu that permits operations such as opening and closing and moving the window.

INTERLISP-D

The version of LISP that has been available on Xerox LISP machines is a special version of INTERLISP, a dialect of the language that goes back to a version of LISP that was first implemented on the PDP-1 by Bolt, Beranek, and Newman (BBN) in 1967. In 1972, the name *INTERLISP* was first used to describe the version of LISP that was implemented as a joint effort of BBN and Xerox PARC. This dialect caught on and resulted in the implementation of INTERLISP on the Xerox Alto in 1974. INTERLISP-D is the dialect of LISP that has grown out of this as the Alto gave rise to a series of custom-microcoded D machines, to which the Daybreak is the most recent addition.

The syntax of INTERLISP-D combines several different constructions, old and new. The way one version of the factorial program looks with this syntax is:

```
(DEFINEQ (FACTORIAL (X)
      (IF (ZEROP X) THEN 1
            ELSE (ITIMES X (FACTORIAL (SUB1 X]
```

```
Loops Icon >
        Dinfo
        Sketch
        VStats
AR Edit  >
FileBrowser
     CHAT
        Idle      >
       SaveVM
Snap
   HardCopy   >
        PSW
        Tedit
   SendMail
  PCEmulation
```

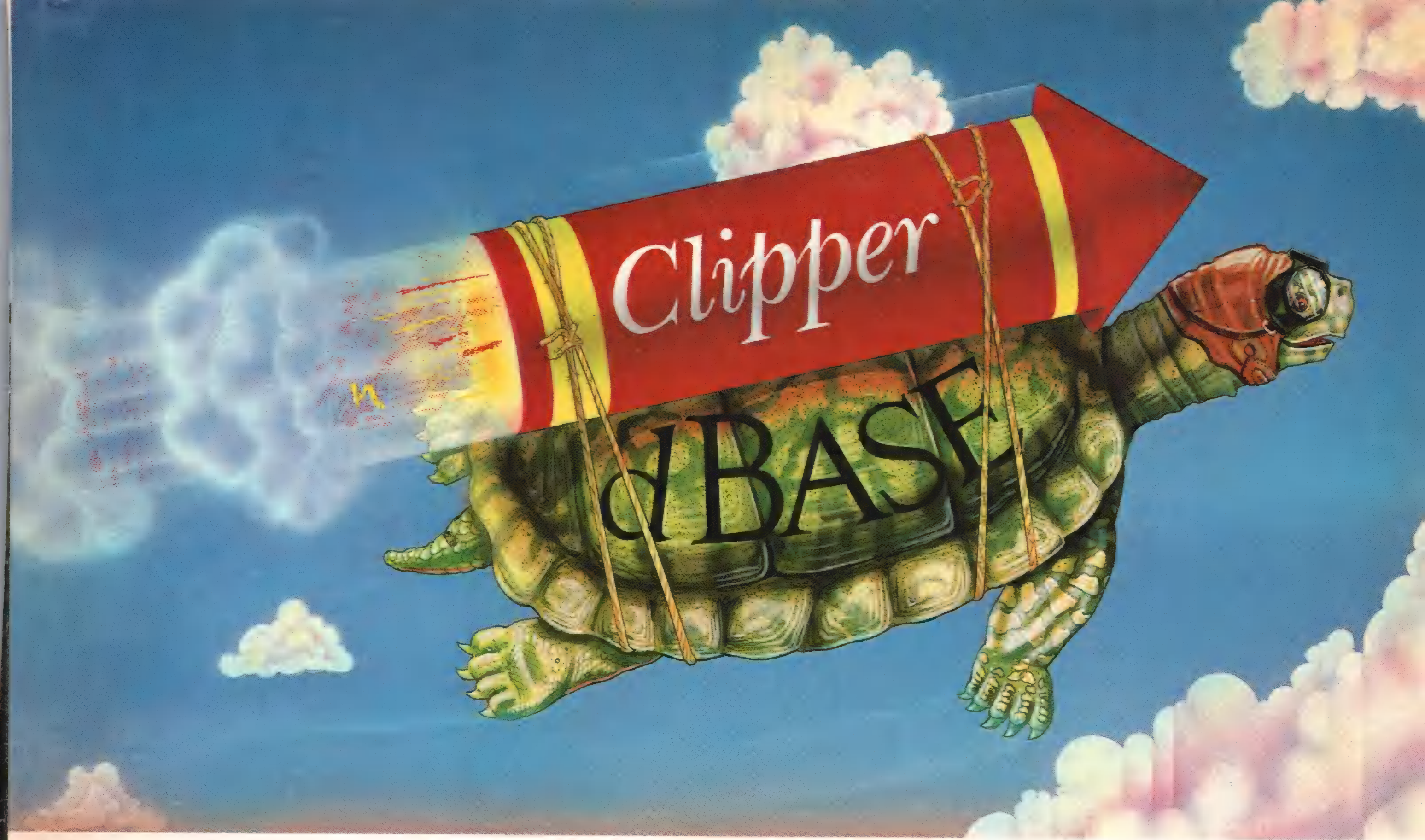
***Example 1:** The Xerox 1186's main menu with LOOPS installed*

Here *ITIMES* is a function for integer multiplication. The value it returns is always an integer; if you pass it a decimal as an argument, it rounds. This function also illustrates the *IF . . . THEN . . . ELSE* macro. You can, of course, use the traditional *COND*, but INTERLISP-D also provides this more-readable syntax for writing conditional tests.

Any time you may need documentation on a function, Dinfo is right nearby. Dinfo is the complete documentation for INTERLISP-D that is available in a flexible on-line facility that includes a graphic tree display of all the topics in the documentation system. Besides being accessible through the graphic browser interface, the detailed documentation can be accessed dynamically for topics as they arise.

Here is a feature I really like: if you begin to enter a LISP function in the Executive, then type a ?, the system looks up the appropriate topic in Dinfo, opens a new window, and displays in it detailed documentation for the function. The graphics browser is also displayed with the current node highlighted so that you can select related topics if you like. This is a remarkably convenient and useful way to provide such elaborate documentation for a programming environment.

By the time this column appears, Common LISP will also be available on the 1186 in its own package or separate namespace. The implementation of Common LISP is a full implementation, developed by extending the kernel of INTERLISP to include all the features of the Common LISP standard. Because of this, statements in INTERLISP and Common LISP can be intermixed freely in applications, and existing programs written in either of the two dialects can be run without making any major changes. Also available for the 1186 is Quintus PROLOG, a standard version of PROLOG making use of microinstructions that allow a processing speed of about 50,000 LIPS (logical instructions per second). The PROLOG implementation is configured in such a way that applications can be written partly in LISP and partly in PROLOG.



Real programmers don't use dBASE. Or do they?

We're finding that some very swift programmers are using it to write some very fast applications, and are completing their projects much more quickly.

Turtle Souped

But they cheat.

They use our Clipper™ compiler to combine dBASE™ with C and assembler.

With dBASE used like pseudo-code, they can then quickly create prototypes that actually run.

Then, with dBASE doing the high-level database functions, they use our Clipper compiler to link in C or assembly language modules from their own bag of tricks.

And they're finding that they're linking in less than they expected because Clipper compiled code runs so fast and because of Clipper's built-in enhancements.

Clipper includes easy networking that provides file and record locking the way it should be done.

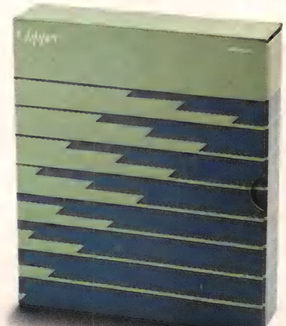
Fast screens that can be treated as memory variables and eliminate the need for direct screen writes and all that tortuous heap management code.

Box commands that make windowing a breeze. And more.

So if you'd like to use your time more productively, check us out: Nantucket Corporation, 12555 W. Jefferson Boulevard, Los Angeles, CA 90066.

Or if you're on deadline, call (213) 390-7923 today.

Clipper could get you out of the soup.



Nantucket®

CIRCLE 220 ON READER SERVICE CARD

ARTIFICIAL INTELLIGENCE
(continued from page 122)

Structure Editors

LISP lends itself well to an editor that knows something about the language. For editing program code on the 1186, you use the SEdit editor, an advanced LISP structure editor. This approach departs significantly from the usual Emacs-like editors used on other LISP machines. A LISP structure editor, as the name suggests, is an editor that knows about the structure of LISP programs in the sense that editing operations are performed on simple or nested LISP expressions rather than on textual structures such as characters, words, lines, and paragraphs. SEdit replaces the earlier DEdit structure editor on the 1100 series machines. Those who have used command-oriented LISP structure editors will probably not realize the enormous difference it makes to use an editor of this kind that is fully integrated with a mouse and window-oriented environment. Once you get the hang of this editor, writing and debugging LISP code is much quicker than with a regular, full-screen text editor.

The SEdit editor differs significantly from the Emacs-like editors used on other LISP machines.

While speaking about editors, I should say something about the Files facility on the 1186. When you use the structure editor to create some code, that code is not just stored in a buffer—it becomes part of the LISP environment. Saving your work to disk, therefore, is not just a matter of writing a text file to disk. It relies on an intelligent facility that keeps track of any changes that have been made to the environment. When it comes time to save anything to disk (functions, variables, windows, bit maps, records, or all of these), you use the same basic procedure. If you say *(FILES?)* in the Exec window, you get a list of variables and functions that are not yet a part of any file. After these are displayed, the system asks "want to say where the above go?". You then type *Y* and are prompted for what to do with each item in turn. An alternate way of ending a session and saving work is provided by the *CLEANUP* command, which handles all this automatically, if it is just a matter of updating already existing files, and compiles all the code as well.

Also in this connection, I should mention the way special editing keys on the keyboard can interact with the mouse to perform powerful operations anywhere in the 1186 environment. An example of this is the COPY key. To use it, you first place the editing caret in the place to which you want to copy the text. You then hold down the COPY key and use the mouse to highlight the text to be copied. You can use this to copy text from any window to any other. One of the most delightful uses of this operation I've experienced on the 1186 is to copy a function right from the editor window into the Exec window, where it becomes immediately available for use without having to be loaded from disk.

Masterscope, DWIM, and Other Power Tools

In the course of developing large programs, particularly ones in which a team of programmers collaborates, it often happens that you forget about various details of functions and variables you have written or you need to know these details about code others have written. Masterscope is a tool that provides several facilities for making it easy to analyze the structure of complex programs. To use Masterscope, you first call upon it to analyze the particular files in which the sources to a program reside. Once you have done this, several commands are available for investigating the code. So, for example, there is the *WHO CALLS* command, which takes the name of a function as an argument. Master-

scope then obediently prints a list of all those functions that call the named function. A related facility, called Databasefns, automatically constructs and maintains Masterscope databases of program files.

DWIM, standing for Do What I Mean, is one of the best-known facilities in the INTERLISP environment. What it does is to try to match unrecognized variable and function names with ones it knows. This amounts to the same thing as a partial match interpreter that is tolerant of misspellings and actually corrects typos on the fly.

Someday we may see an entirely different kind of interpreter that is truly semantically oriented—that would have expectations about what it was going to receive next and that would actively attempt to read input that way and even query the user or programmer to get what it still needed. That would be a really intelligent, forgiving environment—a real DWIM feature—but today all the DWIM we have is tolerance of misspellings.

One of the real delights in the Xerox environment is the SPY window. It is visibly present when the system boots as a large icon of an eye that is tightly closed. If you mouse click on the eye icon, it instantly bursts to life and you see the eye open and freeze in the opened position as if it were looking right at you. The open-eye icon indicates that the SPY facility is active and occupied with keeping track of the time used by various processes.

Conclusions

The Xerox 1186 is an important step toward providing low-cost, dedicated AI workstations. Although the learning curve for getting up and running with any LISP machine should not be underestimated, there are things about the 1186 that make it more accessible than its competitors to new users. Although it took me longer than I expected to gain a working knowledge of the user interface, from the time I've spent working with this machine, I feel it offers a supportive environment for programmers and does what other, more expensive machines do but with noticeably better efficiency. On the other hand, I would not recommend that anyone decide to purchase this machine with the idea of economizing on the peripherals, particularly system memory. The machine I evaluated had about the maximum amount of RAM it can take—3.5 megabytes—and I would not suggest using any less.

Keeping this in mind, I would say that this environment is probably the best buy right now in advanced AI hardware and software technology. Just about all the important higher-end AI tools, such as ART and KEE, run on it. And it is quiet. In spite of their role as personal workstations, many LISP machines are rather noisy.

But perhaps the best reason to get your hands on a Xerox AI workstation is the impressive Xerox LOOPS environment. In my next column I will introduce you to this object-oriented AI programming environment, which Xerox has just made available as a commercial product.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**

(continued from page 12)

mathematics."

Neil D. Pignatano
280 S. Euclid #329
Pasadena, CA 91101

New Wave BASICs?

Dear *DDJ*,

The State of BASIC in the April 1987 issue of *DDJ* was excellent, but the term *new wave* may be misunderstood. Many of the new features cited as included in two specific examples of current BASIC compilers have been available for nearly a decade.

The first new feature touted was alphanumeric lables. Interpreted BASIC language dialects and many BASIC compilers do require line numbers. Microsoft's QuickBASIC supports alphanumeric labels. This doesn't make alphanumeric labels in BASIC new. Digital Research's CBASIC compiler has supported alphanumeric labels for nearly a decade. (I used a copy of Version 1.10, copyright 1981, in the CP/M environment.) So, alphanumeric labels in BASIC have been available for a long time. What is new is the apparent trend toward the use of alphanumeric labels in BASIC compilers.

I loved the reference to alphanumeric-named subroutines with parameter passing as part of this new wave. I classify these subroutines as merely multiple-line, user-defined functions with parameter passing. Again, this is not a new feature in BASIC. CBASIC has supported multiple-line, user-defined functions with parameter passing for a long time. CBASIC also has an assembly-language interface with object file librarian and an overlay linker. New is the trend toward standard BASIC language products that have the features that CBASIC, Pascal, FORTRAN, and C have had for many years.

For those of us who learned FORTRAN as our first language and found Dartmouth BASIC an abomination, CBASIC was a breath of fresh air. There was little that CBASIC could not do, in a structured way, for those of us who started with mainframes and migrated to desktop personal computers running the CP/M operating system in the 1970s. The real surprise to me is why it took Microsoft so long to include the features of CBASIC in a BASIC compiler. It has not yet included all the object file librarian and memory overlay features. Although you can now write multiple-line, user-defined functions, you still can't separately compile them, library them, and link them in at compile time. Perhaps the new wave will eventually catch decade-old CBASIC yet.

The real new wave in BASIC languages is buried by the high-priced advertising of the big software houses. There is an obscure, unadvertised BASIC language with real power. This product is the Minnow Bear BASIC compiler, MB86. It uses the CBASIC language syntax but supports long integers, color, windowing primitives, BCD arithmetic, DOS sytem calls, the 8087, and full 640K memory usage, to cite a few of its features. MB86 compiles to Microsoft C language source, which uses include files for standard routines. This means that those of us who used CBASIC and migrated to C have a BASIC that is full featured and powerful.

MB86 uses the Microsoft C compiler and linker to produce .EXE files under PC-DOS and MS-DOS. Users can write C modules and include them in libraries or directly in the C source generated. The compiled programs are as fast or faster than Digital Research's CBASIC compiler. MB86 has eliminated the memory, 8087 support, and file size limitations of CBASIC. It has virtually all the characteristics of CBASIC without the limitations and is a real contribution to the advanced state of BASIC.

There is ample reason for Microsoft languages to support protected mode under advanced DOS on the 80286 and 80386 processors. Microsoft C should be one of the first compilers written to use ADOS. This means that Minnow Bear BASIC will be able to utilize protected mode shortly after ADOS and the C compiler are available. For some of us, MB86 is the cutting edge of the new wave in BASIC languges. It is far ahead of the products used as examples.

Keith R. Plossl
George Plossl Educational Services
One Parkway 75 Center
1850 Parkway Pl., Ste. 335
Marietta, GA 30067

Still Searching for a Sine

Dear *DDJ*,

I wish to make a comment on the running discussion of the "best" approximation to any given function or collection of data points. This has most recently concerned techniques for approximating the sine function. My comment is that the word *best* should not be applied to any method independently of the context within which the method is to be applied. Despite our desire to believe that science and mathematics provide absolute answers to such questions, pragmatic and even subjective values arise regularly.

When doing a linear least-squares fit to empirical data, for example, we compute the slope and intercept of a straight line that is chosen to minimize a certain sum. Each term in the summation is the squared difference between the line ordinate at an observed abscissa and the corresponding observed ordinate. Why is this quantity minimized? The answer is no less subjective than the answer to the question of why a straight line is used in the fit instead of a parabola or some other function. But minimizing the sum of the squared deviations has become so standardized we seldom ask whether it is the "best" way, and that is why it has become a standard. It is usually the best way because the sum involved is positive, definite, and easily differentiated (at least in the case of polynomial fits), so the minimization conditions are easy to compute. Besides these practical considerations, what we are really doing when we use least-squares fits is we are claiming that the price of being wrong in our approximation is proportional to the sum of the squared deviations from it.

This is usually a good cost function to use, but we could use the sum of the cubed deviations, or the fourth power of the deviations, or many other functions of the deviations. Note that the cubed deviations are a poor choice, as are all odd-numbered powers, because negative deviations contribute negative cost. Another advantage of the least-squares approach is that we obtain the actual

value of the root-mean-square deviation of the fit quite easily, and we have come to regard this as a good figure of merit for judging the quality of the fit. Other cost functions may yield the RMS deviation as easily.

Why use these things called Chebyshev polynomials, then? The answer lies in what their cost function is: the maximum absolute deviation of the fit. In general, Chebyshev polynomials yield higher RMS deviations, but this is often a small price to pay to minimize the maximum absolute deviations. They are also slightly more difficult to compute than least-squares polynomials but not enough to make this a significant consideration. When designing an algorithm to be used on a computer, we are dealing with limited precision; if we can find an algorithm with a maximum absolute error below the precision cutoff, then we know that no better accuracy can be attained for the given precision. This is not as easy to determine when all we know is that the RMS deviation has been minimized; there may be a range in which the absolute error is shockingly high, with excellent behavior elsewhere masking the weakness. This also points out the necessity for great care in selecting points to fit and ranges over which to apply a single fit.

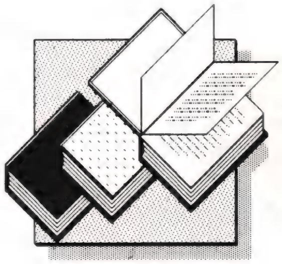
So far, this addresses only numerical accuracy as a consideration. There is also the question of computational speed. Sometimes very rough answers are all that are needed, but speed is critical.

There are many more methods, cost functions to minimize, subjective aspects to be weighed, and so on. I hope only to have stimulated some ideas that may prove useful in determining what is best for the problem you are working on at the moment.

John W. Fowler
Global Solutions
230 Pacific St., #205
Santa Monica, CA 90405

DDJ

BOOKS



Hillis, W. Daniel. *The Connection Machine*. Cambridge, Mass.: MIT Press, 1985.

Imagine a computer that could change its internal structure to handle different types of problems—a massively parallel computer whose data paths could be configured to form a tree structure for finding maxima or minima or a directed graph for solving "shortest path" problems. Daniel Hillis imagined such a beast, and now, as president of Thinking Machines Corp., he is building it. But before he was a company president, he was a student at MIT, and the Ph.D. thesis he wrote there has been published in book form as *The Connection Machine*.

In this book, Hillis explains why conventional von Neumann architectures are inadequate. He graphically illustrates the "von Neumann bottleneck" by asking you to visualize all the elements of a modern mainframe computer on a single piece of silicon. This "megachip" would be a square meter in size and would contain about 1 billion transistors. The CPU, however, would take up only 2 or 3 square centimeters.

Hillis proposes a computer architecture based on two requirements: parallel processing and a dynamically configurable communications network between the processors. He outlines the structure of the CM-1, a prototype machine with 65,535 nodes, each containing its own processor and 4,096 bits of memory. The nodes are coupled by special routing hardware that allows messages to be passed from processor to processor.

Hillis next describes how the machine is programmed. It is connected to a conventional mainframe or supermini that can write directly to the memory of the Connection Machine. A version of LISP running on the host computer causes data and instructions to be passed to the Connection Machine. Hillis describes three extensions to the LISP language that form the basis for exploiting the parallelism of the Connection Machine architecture. He examines the implementation of the architecture in detail and follows this with a description of how various algorithms and data structures can be adapted to the machine.

One of this book's greatest strengths is that Hillis brings his vision to the reader in a clear and logical manner. From an abstract discussion of the need for parallel architectures to the specifics of the CM-1, he covers his topic thoroughly. Although you do not have to be an expert to understand the book, you will probably need some knowledge of computer architecture, algorithms, and LISP. I have no hesitation in recommending it to anyone interested in the future of computer architecture.

Stroustrup, Bjarne. *The C++ Programming Language*. Reading, Mass.: Addison-Wesley, 1986.

The computer science research community has been experimenting with object-oriented programming languages (OOPLs) for more than 15 years, but the benefits of working with these languages have only recently been made available to commercial programmers. Unfortunately, most OOPLs have been interpretive in nature; thus, they have served well as design and prototyping tools, but they have not generally been appropriate for marketable, end-user applications. *The C++ Programming Language* introduces you to a compiled OOPL that was designed to combine the numerous advantages of object-oriented programming style with the efficiency of a compiled language. This book was written by Bjarne Stroustrup of Bell Labs, who designed C++ and its predecessor, C with Classes. The language has roots in both C and Simula67.

The structure of the book is similar in many ways to the classic C reference by Kernighan and Ritchie. It even begins with C++ variants of the "Hello, world" and "English-to-metric" conversion programs. Readers familiar with C may find these programs and several other sample programs frustrating because they do not illustrate object-oriented programming at all. Their inclusion can be justified on the grounds that programmers approaching C++ for the first time will need to learn the standard forms for looping, data structuring, and so on. I think, however, that examples of the object paradigm are just as important as the language syntax.

It is not until Chapter 5 that Stroustrup really begins to discuss the most important feature of C++—the *class* mechanism. From this point on, the book covers object programming in C++ in detail. The text and examples illustrate the features of C++ that take it well beyond C, including information hiding, inheritance, and operator overloading. Stroustrup is also good at pointing out idiosyncrasies in the language that are liable to be misunderstood. Unfortunately, in his example programs, he uses some shorthand notations that I feel are inappropriate. He frequently substitutes the keyword *struct* for *class { public:* when defining classes. This is correct, but it can lead to confusion over what is an object vs. what is a simple data structure.

As a reference manual, this book is indispensable. You should not purchase this book as an introduction to object-oriented programming, however. Stroustrup does not cover the rationale for any of the design decisions or contrast the implementation of C++ with any other OOPLs. Yet without providing this background, he expects readers to recognize that procedure calls (with objects as parameters) replace message passing and that all message-to-object binding is done at compile time via function prototyping. If C++ becomes as popular as C is now, however, *The C++ Programming Language* will be popular indeed.

— Ross Nelson

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 9.**

USE THE BRAINS YOUR IBM WASN'T BORN WITH.

Right at your fingertips in CompuServe's IBM® Forums.

Our IBM Forums involve thousands of users worldwide who will show you just how easy it is to get the most from your IBM and IBM compatibles.

The IBM New Users Forum lets you ask basic questions of PC experts. The IBM Junior Forum is perfect for PCjr® users. Trade tips with other IBM PC and AT users in the IBM Software Forum. Ask questions and get answers directly from the manufacturers in the PC Vendor Support Forum. And if you're looking for a PC Bulletin Board, visit the IBM Communications Forum. Or try the IBM Hardware Forum for discussions on hardware topics and product updates.

Easy access to free software, including free uploads.

You can easily download first-rate, non-commercial software and utility programs. Upload your own programs free of connect time charges. And take advantage of CompuServe's inexpensive weeknight and weekend rates, when forums are most active and standard online charges are just 10¢ a minute. You can go online in most areas with a local phone call. Plus, you'll receive a $25.00 Introductory Usage Credit when you purchase your CompuServe Subscription Kit.

Information you just can't find anywhere else.

Use the Forum Message Board to send and receive electronic messages. Join ongoing, real-time discussions in a Forum Conference. Communicate with industry experts, including the programmers who write your favorite programs. Search Forum Data Libraries for non-commercial software and shareware.

Enjoy other useful services too, like electronic editions of popular computer magazines.

All you need is your IBM computer or IBM compatible computer (or almost any other personal computer) and a modem.

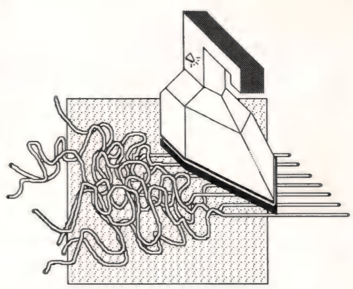
To buy your Subscription Kit, see your nearest computer dealer. Suggested retail price is $39.95. To receive our free brochure, or to order direct, call 800-848-8199 (in Ohio and Canada, call 614-457-0802). If you're already a CompuServe subscriber, type GO IBMNET (the IBM Users Network) at any ! prompt to see what you've been missing.

CompuServe®

Information Services, P.O. Box 20212
5000 Arlington Centre Blvd., Columbus, Ohio 43220
800-848-8199
In Ohio, call 614-457-0802
An H&R Block Company

CIRCLE 237 ON READER SERVICE CARD

THE STATE OF BASIC



Fundamental Data Types in the New BASICs

Microsoft has followed a relatively consistent scheme in implementing data types in MS-BASIC, BASICA, and many other microcomputer BASIC dialects. In MS-BASIC and BASICA, the types supported are integer, floating point, double-precision floating point, and string. Characters are considered to be strings containing a single character. You can associate the data type with a BASIC variable in two ways: use a symbol at the end of the variable name to explicitly indicate the type, or use a *DEFxxx* declaration to perform implicit type declaration.

QuickBASIC uses exactly the same data typing method as BASICA and MS-BASIC do. QuickBASIC strings can be as large as 32K in size, however.

Turbo BASIC follows the same scheme but adds two new items: long integers and integer constants. Long integers support a range between minus and plus two billion. The symbol used for long integers is the & character. Named constants in Turbo BASIC are integer-type identifiers that begin with the % character. For example:

```
%Max.Size = 100
```

defines a named constant *Max.Size* and assigns it a fixed value of 100. Named constants are useful for performing changes in a program without hunting for specific numbers (an operation that sometimes can be hazardous).

Other BASIC implementations are able to mimic named constants by using functions that return a value. For example, in QuickBASIC you can define the following function:

```
DEF FNMax.Size% = 100
```

Although named constants can be faster than function calls, when you use the function call technique, you can simulate constants that are of other types, such as:

```
DEF FNActor$ = "Don Johnson"
DEF FNWeekly.Salary! = 150000.00
```

True BASIC takes an entirely different approach—it supports both numbers and strings. The internal storage format for numeric variables uses a variant of integers and floating points. As long as the number has no fractional part, it is stored internally as an integer. Add a fraction, and the number is stored as a real:

```
LET N = 0 ! N is stored as an integer
LET N = SQR(N) ! N is stored as real
```

Variables and functions in True BASIC are numeric if their names do not end with a $ character. The $ sign is the only data type symbol True BASIC uses—it does not use the implicit *DEFxxx* declaration. Strings in True BASIC can be 64K long.

BetterBASIC uses a Pascal-like approach, declaring the data types of variables and supporting new data types. You declare data types by first stating the data type and then listing the variable name as in:

```
INTEGER: Count, Size, Height
REAL: Salary, Interest
STRING: Name, Message
```

BetterBASIC supports three types of numeric data types: *BYTE*, *INTEGER*, and *REAL*. The *BYTE* type uses 1 byte of storage and offers a range of values between 0 and 255. The *INTEGER* type requires 2 bytes of storage and offers the traditional integer range. The *REAL* type in Better *BASIC* ranges from about 1E+254 to 1E–255, with a user-assigned accuracy.

Strings in BetterBASIC have a default size of 16 characters. They can be as large as 32K and can be allocated as static or as dynamic variables. You can even use the extended memory space to store strings. Consider the following declaration:

```
STRING: Name, ThisLine[80], Buffer/
                    X[3000], AnyString[?]
```

It declares *Name* as a string of default size, *ThisLine* as a string of 80 characters, *Buffer* as a string of 3,000 characters (stored in the extended memory), and *AnyString* as a dynamic string. BetterBASIC also supports the pointer data type. Although its use with fundamental types is somewhat limited, pointers shine when used with record structures, also implemented in BetterBASIC.

BetterBASIC supports named constants that can be of any valid data type. You declare them using the keyword *CONSTANT* followed by one or more constant definitions, such as:

```
CONSTANT Actor$ ="Tom Selleck",
              Series="Magnum, P.I."
CONSTANT ThisNumber=123, That-
                          One = 1.234
```

As you can see from this example, BetterBASIC is able to deduce the type. Although the constant *Actor$* is explicitly typed with the $ sign, the constant *Series* is not, and neither are constants *ThisNumber* or *ThatOne*. BetterBASIC can deduce that *Series* is a string constant, *ThisNumber* is an integer constant, and *ThatOne* is a floating-point constant. If you assign a number to a named constant and the number contains a decimal or an exponential or is outside the range of integers, the constant is associated with a floating-point type. Otherwise, BetterBASIC assumes that you are declaring an integer-named constant.

The new BASICs offer more diversity in data typing than do the first generation of microcomputer BASICs. At the two ends of the spectrum are BetterBASIC and True BASIC, which offer sophisticated and simple data typing, respectively. Microsoft has elected to make QuickBASIC keep the traditional types of BASICA and MS-BASIC. Turbo BASIC has extended some of the BASICA data types.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 10.**

PMI is now the leading international supplier of tools for the language that developers of large projects prefer:

MODULA-2

Our Products Include:

★ ***Repertoire™:*** the largest and most successful library for M2. Includes display system for windows, forms, help, menus, etc. *(MS Windows compatible);* expression evaluation tools; DBMS with variable-length keyed records; text editor; extensive DOS & BIOS access; 330 page manual; full source code (over 600K)................. $89

NEW: VOICE INPUT

★ ***AVOS:*** a price breakthrough in voice-recognition subsystems, complete with expansion board, all necessary hardware, and full Modula-2 source code (including ***Repertoire***); **Comprehends Continuous Speech;** telephone interface available; free AVOS manual provides historic opportunity to learn. $2200

★ ***EmsStorage:*** high-level storage module with full garbage collection and MS Windows-like memory management; uses LIM expanded memory if present, DOS memory if not. $49

★ ***Graphix:*** the only Modula interface to MetaWINDOW, the professional graphics system PCTJ named 7/85 Product of the Month; includes full MetaWINDOW package.................... $149

★ ***ModBase:*** a B+Tree DBMS that uses a file format compatible with Ashton-Tate's dBase III. Includes full source code....... $89

★ ***Macro2:*** a macro preprocessor for Modula-2; provides inline expansion of functions, include files, conditional compilation, etc.
With full source: $89
object-code only:............. $49

*All available exclusively from PMI; dealer inquiries welcome. **Full documentation for all products available free of charge.***

PMI

VISA/MC
AMEX/COD/PO

(503) 777-8844
BIX: pmi
CIS: 74706,262

4536 SE 50th
Portland, OR 97206

CIRCLE 239 ON READER SERVICE CARD

The Advanced Programmer's Editor That Doesn't Waste Your Time

EPSILON

- Fast, EMACS-style commands—completely reconfigurable
- Run other programs without stopping Epsilon—concurrently!
- C Language support—fix errors while your compiler runs
- Powerful extension language
- Multiple windows, files
- Unlimited file size, line length
- 30 day money-back guarantee
- Great on-line help system
- Regular Expression search
- Supports large displays
- Not copy protected

Only $195

Lugaru Software Ltd.
5740 Darlington Road
Pittsburgh, PA 15217

Call (412) 421-5911

for IBM PC/XT/AT's or compatibles

CIRCLE 135 ON READER SERVICE CARD

USERS CAN ADD NEW FUNCTIONS TO LOTUS, WORDSTAR AND dBASE

Move into

THE WEINER SHELL

GET ROOM TO GROW

- Add your own functions to LOTUS, WORDSTAR, dBASE & most other programs on the market.
- Custom-design memory-resident windows, menus, screens & utilities.
- Run Shell programs without leaving your application.
- Execute Shell programs automatically, at timed intervals, or with user-defined hot keys.
- 50K memory required. Supports up to 8M bytes of Lotus-Intel memory.
- $199 (includes limited no-royalty agreement).

THE FIRST MEMORY-RESIDENT PROGRAMMING LANGUAGE

GRYPHON microproducts

P.O. Box 6543/Silver Spring, MD 20906/(301) 384-6868

CIRCLE 374 ON READER SERVICE CARD

The fastest C

Your search for execution speed is over. The new Microsoft® C Compiler Version 4.0 is here. With blazing performance. We've added common sub-expression elimination to our optimizer that produces code that rips through the benchmarks faster than ever before.

"...the Microsoft performance in the benchmarks for program execution is the best of the lot overall." —William Hunt, *PC Tech Journal*, January, 1986.*

But speed isn't the only edge you get with Microsoft C. Other advantages include a variety of memory models like our new HUGE model that breaks the 64K limit on single data items. Plus our NEAR, FAR and HUGE pointers, which provide you greater flexibility. All this allows you to fine tune your program to be as small and fast as possible.

"Excellent execution times, the fastest register sieve, and the best documentation in this review ...Microsoft Corporation has produced a tremendously useful compiler." —Christopher Skelly, *Computer Language*, February, 1986.

No more debugging hassles.
Introducing CodeView. Free.

Now, for a limited time, we'll give you an unprecedented programming tool when you buy Microsoft C, free. New Microsoft CodeView™ offers the most powerful tool yet in the war on C bugs. Forget the hex dumps. Now you can view and work with programs at any level you want. Use the program source, the disassembled object code, or

Microsoft C Compiler Version 4.00

Microsoft C Compiler
- Produces fast executables and optimized code including elimination of common sub-expressions. NEW!
- Implements register variables.
- Small, Medium and Large Memory model libraries.
- Compact and HUGE memory model libraries. NEW!
- Can mix models with NEAR, FAR and the new HUGE pointers.
- Transport source and object code between MS-DOS® and XENIX® operating systems.
- Library routines implement most of UNIX™ System V C library.
- Start-up source code to help create ROMable code. NEW!
- Full proposed ANSI C library support (except clock). NEW!
- Large number of third party support libraries available.
- Choose from three math libraries and generate in-line 8087/80287 instructions or floating point calls:
  - floating point emulator (utilizes 8087/80287 if installed).
  - 8087/80287 coprocessor support.
  - alternate math package – extra speed without an 8087/80287.
- Link your C routines with Microsoft FORTRAN (version 3.3 or higher), Microsoft Pascal (version 3.3 or higher) or Microsoft Macro Assembler.
- Microsoft Windows support and MS-DOS 3.1 networking support.
- Supports MS-DOS pathnames and input/output redirection.

Microsoft Program Maintenance Utility. NEW!
- Rebuilds your applications after your source files have changed.
- Supports macro definitions and inference rules.

Other Utilities
- Library Manager.
- Object Code Linker.
- EXE File Compression Utility.
- EXE File Header Utility.

C Benchmarks In seconds

	Microsoft C 4.0	Lattice C 3.0	Computer Innovation C 2.3	Aztec C86 3.2	Wizard C 3.0
Sieve of Eratosthenes (register)	82.9	151.4	172.3	88.0	91.9
Copy Block	86.9	231.7	199.0	123.8	189.5

Run on an IBM PC XT with 512K memory

Microsoft CodeView
Window-oriented source-level debugger. NEW!
- Watch the values of your local and global variables and expressions as you debug.
- Set conditional breakpoints on variables, expressions or memory; trace and single step.
- Watch CPU registers and flags as you execute.
- Effectively uses up to four windows.
- Debug using your original source code, the resulting disassembly or both intermingled.
- Use drop-down menus to execute CodeView commands.
- Access the on-line help to lead you through CodeView's options and settings.
- Easily debug graphics-oriented programs since program output is kept separate from debugger output.
- Keyboard or optional mouse support.
- Enter in familiar SYMDEB or DEBUG commands.

you've ever seen.

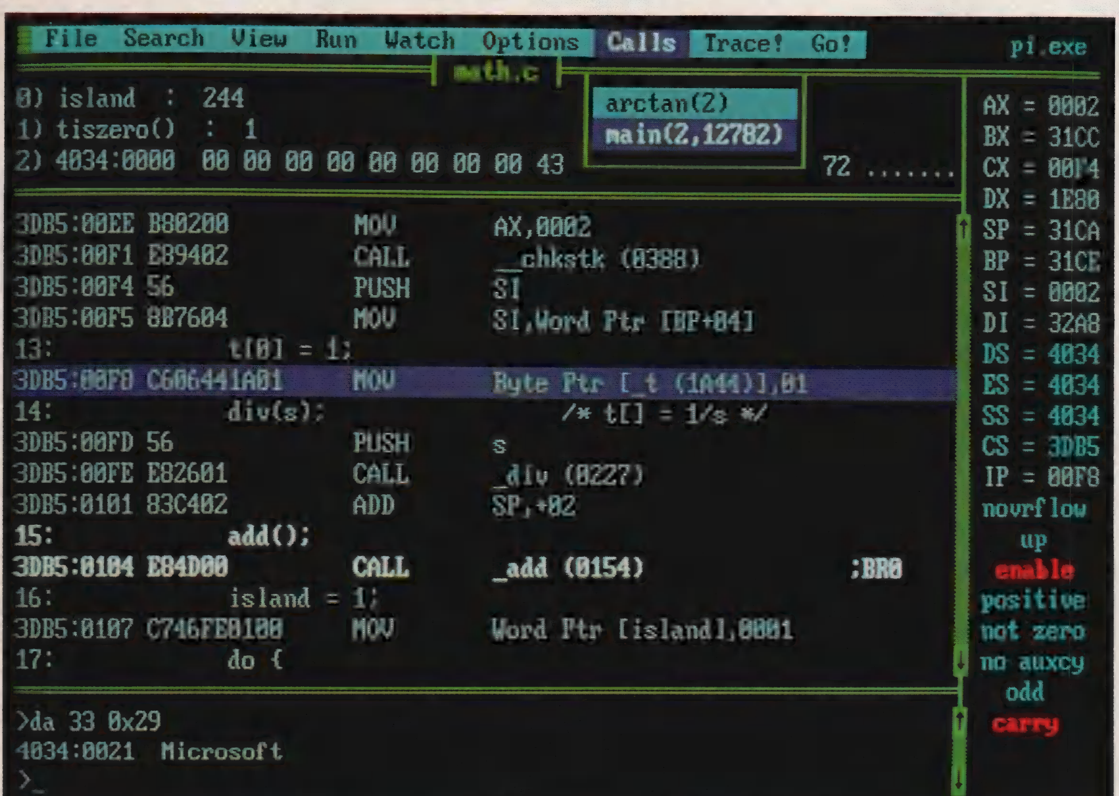
both at the same time. Open a window to view CPU registers and flags. Watch local and global variables as well. All while your program is running.

CodeView gives you complete control. Trace execution a line at a time—using source or assembly code. Or set conditional breakpoints on variables, memory or expressions. CodeView supports the familiar SYMDEB command syntax, as you'd expect. Commands are also available through drop-down menus. Combine the new window-oriented interface with our on-line help and debugging has never been easier. Or quicker.

Take the $5 CodeView tour.

You may find it hard to believe our debugger can do all we've claimed. So we're offering test drives. Five bucks will put you behind the wheel of a Microsoft C demo disk with CodeView.† See for yourself how fast debugging can get.

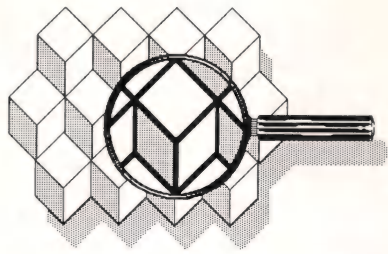
For more information about the CodeView demo disk, the new Microsoft C Compiler, a list of third party library support or the name of your nearest Microsoft dealer, call (800) 426-9400. In Washington State and Alaska, (206) 882-8088. In Canada call (416) 673-7638.



Microsoft® C Compiler

The High Performance Software

OF INTEREST



Miscellaneous

Computer Professionals for Social Responsibility is sponsoring a symposium entitled Directions and Implications of Advanced Computing, in Seattle, Washington, on July 12. The aim of the symposium is to consider the directions and implications of advanced computing in a social and political context as well as a technical one. Symposium topics will include computing research funding, defense applications, computing in a democratic society, and computers in the public interest. Keynote speakers will be Robert Kahn, formerly director of the Information Processing Techniques Office at the Defense Advanced Research Projects Agency, and Terry Winograd, an associate professor of computer science at Stanford and an AI maven. Proceedings will be distributed at the symposium and will be on sale during the 1987 AAAI conference. Reader Service No. 29.
Computer Professionals for Social Responsibility
P.O. Box 85481
Seattle, WA 98105
(206) 783-0145
(206) 548-4117

Flambeaux Software has announced the availability of TECH Help!—The Electronic Manual, an on-line, pop-up, technical reference manual of the most commonly needed information for system-level programmers. It includes comprehensive coverage of the DOS and ROM BIOS services; system variables; I/O ports; installable device drivers; display usage (including the EGA); and the layouts and structures of dozens of data tables, bit flags, and switch settings. It is up to date, covering topics through DOS 3.2 and the latest PC/AT BIOS. It also describes the Lotus/Intel/Microsoft Expanded Memory Specification. TECH Help! pops up from within your program editor or debugger to give you instant access. It sells for $69.95. Reader Service No. 30.
Flambeaux Software
1147 E. Broadway, Ste. 56
Glendale, CA 91205
(818) 500-0044

Connections is a bimonthly newsletter for networked Macintoshes, designed to provide answers about network products and planning for both novice and experienced network users. It covers topics of interest to Macintosh users who wish to exchange data and information among themselves as well as with users of other kinds of computers. Future issues will include articles such as IBM 3270 connectivity, Unix connectivity, file servers, star controllers, file transfers and conversions, AppleTalk utilities and diagnostics, and the use of other file transfer protocols on the Macintosh. A one-year subscription to *Connections* costs $60 ($70 for overseas subscribers). Reader Service No. 31.
David R. Kosiur
Connections
P.O. Box 5894
Fullerton, CA 92635
(714) 738-1492

The Visible Computer (TVC):8088, from **Software Masters**, is a book and disk combination for mastering 8088 assembly language. It consists of a 350-page, tutorial-style manual, a program that graphically simulates the inner workings of the 8088 chip, and dozens of demonstration programs. *TVC* is designed for people with no prior exposure to assembly language and includes preliminary chapters on hex and binary numbering systems. *TVC:8088* for PC-DOS machines requires 128K RAM, is not copy-protected, and sells for $79.95. Reader Service No. 32.
Software Masters
P.O. Box 3638
Bryan, TX 77805
(409) 822-9490

Novation has introduced a 300/1,200-baud, AT (Hayes)-compatible modem called the Parrot 1200. The modem is approximately the size of an audio cassette (4¼ × 2¾ × ⅝ inches) and weighs three ounces. A microprocessor-controlled power-management system enables the Parrot 1200 to function at high levels of reliability using only the power available from the host computer's RS-232 serial interface; neither batteries nor external AC power are required. Features include transmission speeds of 0–300 or 1,200 bps; Bell 103/212A hardware compatibility; an asynchronous data format; full-duplex operation; built-in auto self-test, analog loop-back, local digital loop-back, and remote digital loop-back testing; a speaker with volume control; four LED indicators; and an AT-standard (Hayes) command format. The Parrot 1200 sells for $119. Reader Service No. 33.
Novation Inc.
21345 Lassen St.
Chatsworth, CA 91311
(818) 988-5060

A six-volume set of books that serves as a software management tool for establishing a company's internal programming and documentation practices is available from **ATC Software**. Five of the volumes describe standard methods for programming in COBOL, FORTRAN, C, BASIC, and dBASE. The sixth volume describes uniform software documentation standards for the five languages. The set sells for $72; individual volumes cost $15 each. Reader Service No. 34.
ATC Software
Rte. 2 Box 448
Estill Springs, TN 37330
(615) 967-9159

DDJ

USING THE WRONG LANGUAGE CAN BE MURDER. SPEAK SMALLTALK/V.

Let's talk languages. Programming languages like Turbo Pascal, C or Basic can be killers. To many, they're foreign, complex, and generally intimidating. Mistakes can be deadly.

With Smalltalk/V, you have an elegantly simple solution that puts the power and majesty of a major AI programming language on your PC or compatible. It makes no difference if you're an experienced programmer or just getting started. Smalltalk/V gives you an easy-to-use and flexible programming tool.

This is the same language used by leading software companies for their new product development. There are sound reasons for this. Smalltalk/V offers a totally integrated programming environment using the premier object-oriented language. You use natural language rather than complex programming codes. It puts Macintosh-type graphic features on a PC including overlapping windows, bit-mapping, pop-up menus, and a mouse interface. More than mere window dressing, Smalltalk/V delivers fully interactive windows that are easy to build and quick to modify.

But don't just take our word on it. Hear what the experts have to say:

"This is the real thing folks. A super Smalltalk like this turns your PC into a hot workstation. It's fantastic... Highly recommended."

John Dvorak
Contributing Editor
PC Magazine

"The tutorial provides the best introduction to Smalltalk available."

Dr. Andrew Bernat
AI Expert Magazine

"Smalltalk/V is the highest performance object-oriented programming system available for PCs."

Dr. Piero Scaruffi
Chief Scientist
Olivetti Artificial
Intelligence Center

Today, thousands of professionals, scientists and engineers are using Smalltalk/V to solve both simple and expert problems. Giving them a new dimension in computer applications for their PC.

Put new life into your PC by calling toll free 1-800-922-8255 and ordering Smalltalk/V today. Smalltalk/V by Digitalk, Inc., 9841 Airport Blvd., Los Angeles, CA 90045. (213) 645-1082.

$99.95

Smalltalk/V comes with 10 starter applications including Prolog and each Application Pack adds several more. All source code is included. Supports 640 x 480 color graphics with color extension pack.

Smalltalk/V requires DOS and 512K RAM on IBM PC/AT/PS or compatibles and a CGA, EGA, Toshiba T3100, Hercules, or AT&T 6300 graphic controller. A Microsoft or compatible mouse is recommended. Not copy protected.

Turbo Pascal is a trademark of Borland Internation. IBM, IBM PC/AT/PS are trademarks of International Business Machines Corporation. Macintosh is a trademark of Apple Computer, Inc.

TO ORDER CALL 1-800-922-8255 TODAY.

60-DAY MONEY-BACK GUARANTEE*

Send check, money order, or credit card information to: Digitalk, Inc., 9841 Airport Boulevard, Los Angeles, CA 90045.

Credit Card ☐ VISA ☐ Mastercard

Card number: ____________________

Expiration date: ____________________

Name: ____________________

Street Address: ____________________

City/State/Zip: ____________________

Smalltalk/V	$99.95
RS-232 Communications Application Pack	$49.95
EGA/VGA Color Extension Pack	$49.95
"Goodies" Application Pack	$49.95
SPECIAL OFFER: Smalltalk/V and all 3 packs only	$199.95
Shipping and handling (Outside North America	$5.00 $15.00)
California residents add applicable sales tax	
TOTAL	$______

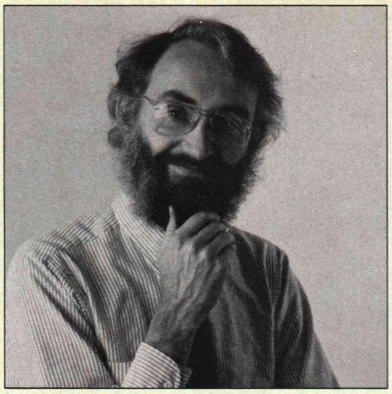
*Unconditional 60-day money-back guarantee. Simply return to Digitalk, Inc. and your refund will be immediately forwarded to you.

Smalltalk/V

digitalk inc.

CIRCLE 127 ON READER SERVICE CARD

SWAINE'S FLAMES



One company worth watching just now is Phoenix Technologies, developer of an operating system extension called VP/ix, a virtual PC environment that may have a lot to say about the popularity of Unix on 386 machines. Phoenix is working with both Microsoft and Interactive Systems to make Xenix and Unix support DOS applications on 386 systems. This puts Phoenix in the heart of Unix development for the 386 because AT&T-Intel-Interactive Systems Unix and Microsoft-Santa Cruz Operation Xenix are the most important strands in Unix development for the 386. Phoenix is especially worth watching because these two strands are converging as a result of this spring's agreement between AT&T and Microsoft. Under that agreement, Microsoft will develop the next version of Unix for AT&T, a version that will be designed for the 386 and that will be upwardly compatible with AT&T's and Microsoft's existing Unix or Xenix products.

Some text is meant only for human processing. What you say in on-line conferences, for example, is often of no lasting import and does not need to be processed in any other way. ROAM, my cousin Corbett calls it: Read Once, At Most. Also, you may prefer not to have your words counted, indexed, or munged without your approval.

Corbett has come up with two data-encoding encryption techniques for text that is only meant for human perusal. One nice feature of such techniques is that only the sender needs any special software; the encrypted message is displayed to its intended recipient and the decryption process is performed in his head.

The simplest is the etaoin encoding. This encoding is trivial to implement, can encode in real time during high-speed transmission, reduces data by up to 50 percent (making a 1200-bps transmission effectively 2400 bps), and produces ciphertext more appropriate for human decryption than for machine decryption. Although a program with access to an English dictionary could crack the messages, even this is questionable if the sender constructs the messages to make maximum use of context dependencies and uses uncommon words.

Here's how it works. It maps uppercase and lowercase *e*s into lowercase, and similarly for *t*, *a*, *o*, *i*, *n*, *s*, *h*, *r*, *d*, *l*, and *u*. (These are the most common letters in written English.) Lowercase is used because the ascenders help distinguish the letters at a glance. All other letters are mapped into underlines, all punctuation into periods, and space into space. Numbers are spelled out. Here is a message in etaoin encoding:

i need to _et a _aster _ode_. at least t_el_e hundred _aud.

Not too hard for human decryption. But note that two things make the machine decryption of this harder than you might at first expect: letter-frequency information is not a useful tool for extracting the remaining, low-frequency letters, and, in general, structural words appear intact, but words crucial to the meaning of the message are ravaged. The simplest way to recover these words is by using semantic context and real-world knowledge, exactly the things that people do naturally and programs don't do. The word *modem* in the message, for example, would be hard to recover without reference to the meaning of the entire message.

The more ambitious of the two encodings is 3-Bit English (3BE). This encoding, which has a greater data-reduction efficiency than etaoin, is based on context-dependent confusibility studies and studies of redundancy-reduction in English prose conducted at Matrix Labs in Research Triangle Park in North Carolina. These studies show how to map letters into an eight-character alphabet so as to lose the minimum information at the lexical and phonetic levels. It turns out that phonetically confusible letters do not often appear in identical contexts for the simple reason that this would lead to auditory confusions.

Corbett's encryption technique maps, for example, phonetically similar letters such as *d* and *t* together in such a way as to minimize the possibility of the reader mistaking the whole word. Vowels, which are high-frequency letters, carry little information and can all be mapped into a single symbol. In addition, Corbett hopes to develop a custom font for the recipient that will make it possible to see the character either as a *t* or a *d*, for example. The visual system, accustomed to resolving ambiguities at the letter level using word-level information, will see the character appropriately. This encoding, depending as it does on subliminal cues, language use, and idiom, should be extremely difficult to crack via computer, yet should be readable by any English-speaking person who can squint.

Michael Swaine

Michael Swaine
editor-in-chief

THE ADA® WORLD HAS CHANGED.

VALIDATED

Meridian's AdaVantage™ v2.0 compiler is a complete implementation of the Ada® language that has been validated on the IBM PC/XT, IBM PC/AT, and the Zenith Z-248. Most of the representation clauses and implementation-dependent features are also available including

- ▲pragma pack
- ▲size specification
- ▲task storage size
- ▲fixed point small
- ▲record representation clauses
- ▲address clauses
- ▲package system
- ▲representation attributes
- ▲pragma interface
- ▲unchecked storage deallocation
- ▲unchecked type conversions

The compiler includes a complete set of utilities for managing the Ada program library and all of the standard packages including text_io, system, and calendar.

BEST PRICE PERFORMANCE

The Meridian AdaVantage compiler demonstrates the best price/performance of any PC Ada compiler. The compiler sells for $795 in single quantities and it compiles about 1000 lines per minute on an IBM PC/AT.

In addition to the production compiler, two other versions are available for general training and student use. The AdaTraining™ compiler sells for $395 and is intended for corporate and university educational environments. The AdaStarter™ compiler, priced at $129, incorporates all of the features of the AdaVantage production compiler, with certain limitations on the number of library units and the number of lines per compilation unit allowed. The full price of the AdaStarter compiler can be applied to a later purchase of the AdaVantage production compiler.

ADDITIONAL PRODUCTS

Two optional packages, priced at $50 each, that provide DOS environment support and miscellaneous utility routines are currently available. A source-level debugger and Ada editor will be available this Fall.

CONFIGURATION

The compilers all run in a standard PC configuration with 640K of memory, a hard disk, and DOS v2.1 or higher.

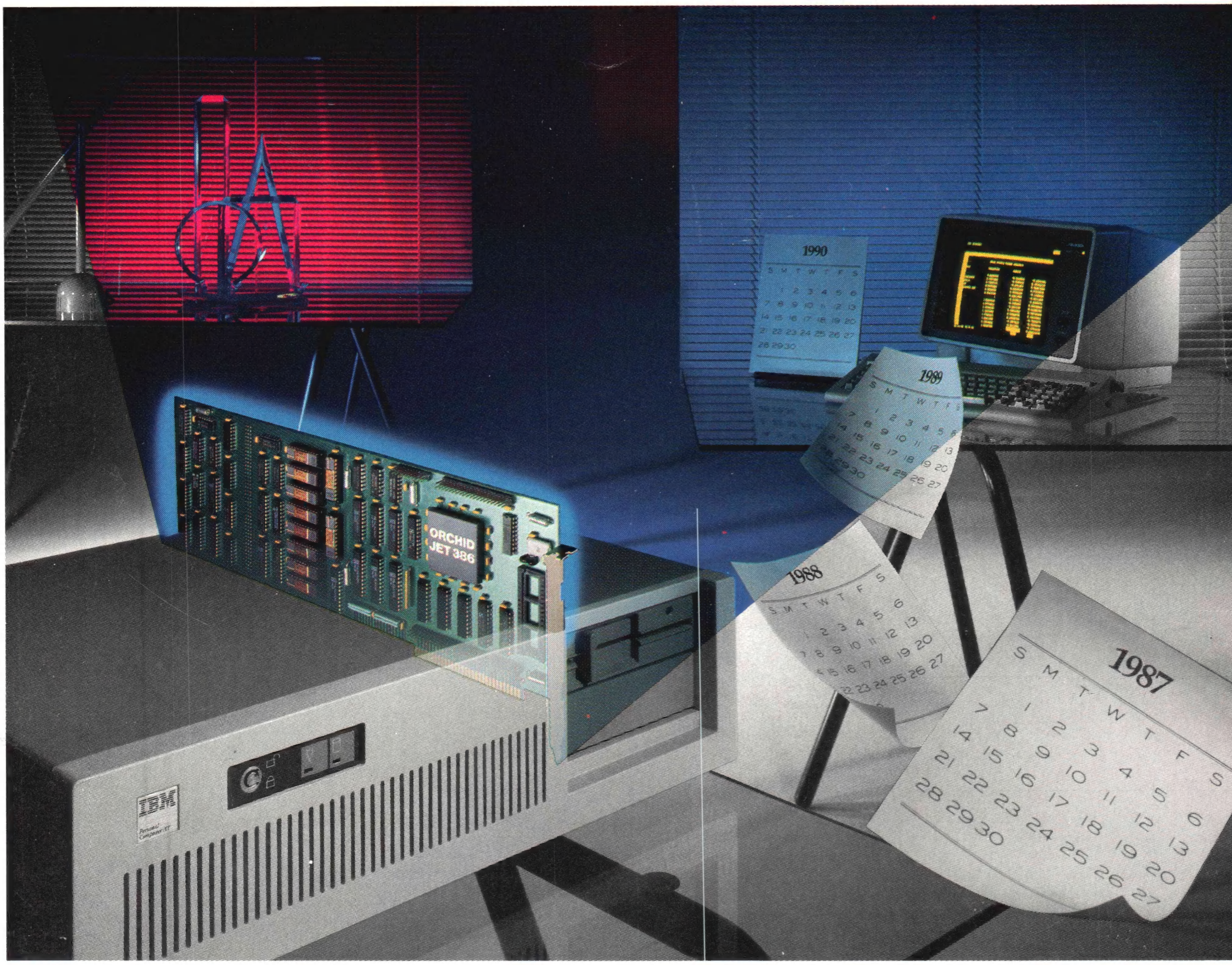
To order today, or get more information, call toll-free 1-800-221-2522.

MERIDIAN
SOFTWARE
SYSTEMS, INC.

23141 Verdugo Drive, Suite 105
Laguna Hills, CA 92653
800/221-2522 (outside Calif.)
714/380-9800 (inside Calif.)
Telex: 650-268-0547 MCI

CIRCLE 350 ON READER SERVICE CARD

ORCHID'S JET 386™: POWER FOR THE FUTURE NOW



Jet 386 is the Ultimate Accelerator Upgrade for Your AT

Announcing an end to obsolescence. Orchid Technology's Jet 386™ accelerator card extends the life of your computer investment into the 1990s—it puts power in your AT that you won't outgrow.

Three Times Faster than an AT

It's up to three times faster than an AT depending on the application, and speed is just one benefit. Unequalled compatibility and provisions for upcoming 386 software mean your Jet 386 will handle whatever the future has in store: CAD, spreadsheets, networking...

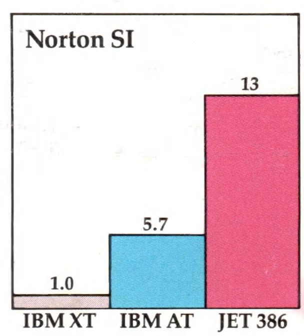
Easy Upgrade

Easy to use, there's nothing new to learn and no new programs to buy. At 25% of the cost of buying a new 386 PC, it's easy on your pocketbook, too.

From the People Who Started It All

Orchid combined 80386 power with the technology perfected for the XT in the TinyTurbo and PCturbo 286e. Like these critically acclaimed accelerators, Jet 386 is built for lasting value.

Call Orchid to find out how you can experience the future today. And ask how Orchid can modernize your whole office with turbos, graphics, networking, and multifunction products.



ORCHID
Innovative Add-Ons

45365 Northport Loop West
Fremont, CA 94538
415/490-8586 Tlx: 709289

CIRCLE 130 ON READER SERVICE CARD